

# **definegeometry2d**

September 10, 2019  
13:04

## **Contents**

<b>1</b>	<b>A program for Defining 2-D Geometries</b>	<b>1</b>
<b>2</b>	<b>Introduction and Background</b>	<b>2</b>
<b>3</b>	<b>Symmetry</b>	<b>4</b>
<b>4</b>	<b>How To Use This Code</b>	<b>5</b>
<b>5</b>	<b>Input File</b>	<b>7</b>
<b>6</b>	<b>Keyword Reference</b>	<b>12</b>
<b>7</b>	<b>Other Considerations</b>	<b>21</b>
<b>8</b>	<b>The Main Program</b>	<b>24</b>
<b>9</b>	<b>Main Subroutine</b>	<b>27</b>
<b>10</b>	<b>Read and process DG file</b>	<b>45</b>
<b>11</b>	<b>Read wall file</b>	<b>51</b>
<b>12</b>	<b>Convert mesh data into polygons</b>	<b>53</b>
<b>13</b>	<b>Set up mesh elements</b>	<b>55</b>
<b>14</b>	<b>Specify and break up a new polygon</b>	<b>58</b>
<b>15</b>	<b>Set up trivial end zones</b>	<b>63</b>
<b>16</b>	<b>Set up toroidally facing sectors</b>	<b>66</b>
<b>17</b>	<b>Set up sector based diagnostics, including those composed of auxiliary strata</b>	<b>69</b>
<b>18</b>	<b>Read mesh file from UEDGE</b>	<b>71</b>
<b>19</b>	<b>Compute minimum and maximum coordinates</b>	<b>72</b>
<b>20</b>	<b>Print wall coordinates to a file for user examination</b>	<b>73</b>
<b>21</b>	<b>Read input lines specifying a polygon</b>	<b>75</b>

<b>22</b> Read data specifying a diagnostic	<b>93</b>
<b>23</b> Fill in data for this zone	<b>96</b>
<b>24</b> Define universal cell	<b>98</b>
<b>25</b> Find the centroid of a triangle	<b>101</b>
<b>26</b> Compute the center of a quadrilateral	<b>103</b>
<b>27</b> Compute area of a closed polygon	<b>103</b>
<b>28</b> Reverse order of polygon points	<b>104</b>
<b>29</b> Find endpoints in a set of elements	<b>105</b>
<b>30</b> Convert elements to a polygon	<b>106</b>
<b>31</b> Count the number of continuous sections in set of elements	<b>108</b>
<b>32</b> Search mesh edges to find the ones corresponding to the input DG label	<b>110</b>
<b>33</b> Attempt to find the closest point on the two input edges to the input point	<b>111</b>
<b>34</b> Find closest point in a set of elements, <i>edge_elements</i> , to the point provided, <i>poly_p</i>	<b>113</b>
<b>35</b> Set up sectors	<b>115</b>
<b>36</b> Print data from Triangle	<b>119</b>
<b>37</b> C interface routine to Triangle	<b>122</b>
<b>38</b> INDEX	<b>127</b>

## 1 A program for Defining 2-D Geometries

This program began as an extension and generalization of the logic used in *readgeometry*. The approach used has been rather successful, permitting a wide variety of geometries to be described in a straightforward manner. While the code can be run using just a human generated text input file, with perhaps some auxiliary data in other text files, truly complicated cases can be handled when the code is run in conjunction with other geometry packages.

For coupling to codes using a structured, quadrilateral mesh, one can take advantage of the interface to the DG code, developed in Garching, that gives the user what is effectively a GUI for the problem at hand. Alternatively, the plasma mesh used by the UEDGE code can be read in via a single command; a plasma mesh in the Sonnet format can be read in a similar manner (Sonnet format grids are also compatible with DG). Unstructured triangular meshes can be transformed into the Sonnet format via the *tri\_to\_sonnet* routine; the input to **definegeometry2d** can then be constructed just as in the quadrilateral mesh case. When the spatial extent of the problem can be restricted to a rectangular region having no intervening material surfaces or X-points, one can use the **efit2dg2d** pre-processor to generate the more tedious parts of the **definegeometry2d** input file.

**definegeometry2d** can now handle nearly symmetric 3-D problems; the essential restriction is that all of the objects in the problem must be describable as 2-D figures swept through a range of the ignorable coordinate.

The intermediate specification of the geometry purely in terms of polygons is written out to a netCDF file. This file can be read in by the *ucd\_plot* post-processor to generate (2-D only) graphics equivalent to those produced by **geomtesta**, but at full resolution. The polygon netCDF file can also be transformed into the **Triangle** format (see below) via the *poly\_to\_tri* routine, providing an additional means of coupling to other applications.

A version of DG has been and is still included in the DEGAS 2 distribution. However, it has been used less frequently and, thus, has not been actively maintained. Users requiring additional features may want to consider the version of DG included with the SOLPS-ITER distribution (called DivGeo there); the version of Carre in that package is also much more advanced than the one in DEGAS 2. See the additional documentation in the DEGAS 2 User's Manual and in the manual provided with DG.

Note that this program makes extensive use of Jonathan Shewchuck's **Triangle** package to break polygons up into triangles. The user needs to download its source code from:

<http://www.cs.cmu.edu/~quake/triangle.html>

and arrange for the **Makefile** to find it (the default location is **\$HOME/Triangle**).

## 2 Introduction and Background

This code takes a human-readable input file that, together with the other file(s) it references, describes a geometry you want to use in DEGAS 2. The purpose of this code is to transform that description into DEGAS 2's internal description of the geometry.

The hierarchy of geometry-related objects in DEGAS 2 is, from the lowest level to the highest level, is:

1. surface,
2. cell,
3. polygon,
4. zone.

Individual cells are composed of surfaces, as discussed in the documentation for the internal geometry (e.g., see *geometry.web*). The next level up (in two dimensions) is a polygon. This program will call various routines to break up (non-convex) polygons into cells. Finally, a zone may consist of one or more polygons; properties are constant across a zone. For example, a single zone might be used to represent the vacuum region around the plasma which is comprised of several (possibly disconnected) polygons. Or, the volume between two plasma flux surfaces in which density and temperature are constant might be a single zone. Note that polygons are used only within external interfaces such as this code. Cells and surfaces are essentially used only by the code itself; the user only deals with them in debugging. Zones are used primarily for the specification of input and output data.

The most common sources of errors in this code are associated with the violation of two basic restrictions on DEGAS 2 geometries:

1. The problem volume (everything inside the “universal cell”) must be completely filled with user-defined zones.
2. Each surface not lying along the boundary of the universal cell must have user-defined zones on both sides of it.

If *definegeometry2d* reports an error, you have most likely violated one (or both) of these two constraints.

Another important component of the geometry are the “sectors”; they are defined at interfaces between adjacent zones of different types to facilitate tracking or between zones of the same type for diagnostic purposes. Flights are temporarily halted at sectors during tracking. If the next zone along the track represents, say, a solid material, the main code hands that flight off to the routines that effect the interaction of the current particle with that material. While stopped at that sector, scores to corresponding “diagnostics” (groups of sectors) are also tabulated. The essential identifying information of a sector are its surface and zone numbers. Currently, sectors are defined on both sides of that interface, one for each of the two adjacent zones. Both will refer to the same surface, although the surfaces will have opposite orientations.

For additional information on sectors see the documentation for the sectors class (in `src/sector.hweb` or `Doc/classes.pdf`) and the “Diagnostic Sectors” section of the User’s Manual.

This code automatically creates sectors at the interfaces between plasma / vacuum zones (i.e., zones in which flights propagate) and solid / exit zones (i.e., zones in which flights do not propagate). A sector between solid and plasma zones is defined to be a “target” sector and one between a solid and vacuum zone is a “wall” sector. The adjoining sectors are referred to as “plasma” and “vacuum” sectors, respectively. For further details, see the sector class described in *sector.hweb*.

Sectors are also labeled by a “stratum” number and a “segment” number. These labels facilitate subsequent user reference to the sectors. Since these two labels are just that, they can in principle be chosen almost arbitrarily (although the choice should be such that a given stratum, segment pair does uniquely identify a sector). Because the process of identifying the default sectors in a problem is based on an examination of all solid and exit polygons, the stratum numbers associated with those polygons are used to label the resulting sectors *on both sides*. Consequently, stratum numbers assigned to vaccum and plasma polygons are currently not used. Default sectors defined at the edges of a *sonnet\_mesh* or *uedge\_mesh* the mesh will carry the stratum label from the adjoining solid polygon.

Each side of a polygon gives gives rise to a different surface. The sector’s “segment” number is derived from the location of that side in the polygon. Again, that segment number is used to label both sectors constructed at that interface.

The user can also specify “auxiliary sectors” that coincide with arbitrary segments of a polygon and even include segments from multiple polygons. The user can then define a diagnostic or diagnostics from either default or auxiliary sectors.

### 3 Symmetry

The `definegeometry2d` code is intended to handle 2-D and nearly symmetric 3-D planar and cylindrical geometries (for a more detailed description of geometry symmetry in DEGAS 2, see the “Symmetry and Coordinate Systems” section of the User’s Manual). In the 2-D cases, one coordinate is ignored ( $y$  in plane symmetry;  $\phi$  in cylindrical symmetry). Tracking is done in 3-D, but the zones constructed from the polygons established in `definegeometry2d` extend the full range of the ignored coordinate, providing no resolution in that direction.

In the nearly symmetric 3-D cases, the 3-D zones are again constructed by sweeping the polygons through the  $y$  or  $\phi$  coordinate. The difference is that those zones are further subdivided by  $y = \text{constant}$  or  $\phi = \text{constant}$  planes. The result is a “bread slice” (in the rectangular case) or a “pie slice” (in the cylindrical configuration) description of the geometry. Consequently, DEGAS 2 results are resolved in that direction. A second difference is that the zone type (solid, plasma, vacuum, or exit) can vary in the  $y$  or  $\phi$  direction. This capability allows some 3-D objects, such as a poloidal limiter in a tokamak, to be simulated.

## 4 How To Use This Code

Virtually all of the information in the `definegeometry2d` input file is “high level”, e.g., point coordinates rarely appear. Instead, most of the detailed information regarding points and their connectivity is contained in the other files that can, and should, be generated by other packages. The names of these files are specified in the first part of the input file. The formats that are currently in use and the corresponding keywords in the input file are (all of these are text files):

*dg\_file* An output (with extension `.dgo`) file from the DG code.

*wallfile* A simple text file containing a specified number of simply connected “walls”, with each wall consisting of a sequence of  $(X, Z)$  values.

*sonnet\_mesh* A 2-D mesh specified in the Sonnet format; e.g., as generated by the `Sonnet` or `Carre` codes.

*uedge\_mesh* A specific format used to transfer data between the `UEDGE` and `DEGAS 2` codes.

Both *dg\_file* and *wallfile* result in the definition of “walls” that can be used to construct the polygons in the second part of the `definegeometry2d` input file using the *wall* keyword. The polygons can also be constructed directly in DG using its graphical interface. These polygons are invoked in the second part of the `definegeometry2d` input file with the *dg\_polygon* keyword. A portion of the boundary of a plasma mesh imported via *sonnet\_mesh* or *uedge\_mesh* can be incorporated into a polygon via the *edge* command. A simpler approach to accomplishing the same result using DG is to associate a “mesh connection” with the polygon; an *iedge* (short for “intelligent edge”) keyword in the `definegeometry2d` input file provides the code with additional information on how to connect the mesh edge to the rest of the polygon. The corners of the universal cell (set with the *bounds* keyword) can be added to a polygon with the *outer* keyword. If points having the exact same coordinates are defined in DG, they can be used there in a *dg\_polygon* and will be recognized as being on the boundary of the universal cell.

The two most frequently used approaches to setting up a tokamak based geometry begin with files such as these generated by the `UEDGE` or DG codes. Typical methodologies for these two cases are:

**UEDGE** A simple geometry can be set up by using the *uedge\_mesh* keyword to establish all of the plasma zones. The *edge* and *outer* commands can be used to define solid zones that enclose this volume. Alternatively, a slightly more complicated approach involves specifying in a *wallfile* a more realistic vacuum vessel shape. The (vacuum) volume between that surface and the `UEDGE` mesh would need to be described by polygons set up using the *wall* and *edge* keywords.

**DG** The 2-D outlines of the vacuum vessel (or other hardware or surfaces) are first defined as “elements” in the DG code. These can subsequently be connected into and used to define DG’s “polygons”. Note that the Carre or Sonnet grid generation codes can be used to create Sonnet format plasma meshes that can be viewed and manipulated (in limited ways) in DG. The edges of the mesh can be incorporated into polygons with “mesh connections”. The DG elements, polygons and other information are contained in DG’s output file. Upon being read into `definegeometry2d`, these objects are reinterpreted in terms of its internal objects. The user can then utilize them in creating the polygons that will divide up the problem space.

Alternatively, there are two possible approaches to generating a new geometry from scratch:

1. If the problem possesses a basic symmetry, e.g., “plasma parameters are constant on a flux surface”, you can specify only those bounding surfaces and let *definegeometry2d* subdivide them (semi-arbitrarily) into smaller zones. First, construct a wall file containing the coordinates of each of the bounding surfaces, with each as a separate wall (see below under the *wallfile* keyword). Then, in the *definegeometry2d* input file, create a polygon out of each pair of adjacent walls, i.e., surfaces. Using the *triangulate\_to\_zones* keyword will then break these polygons up into smaller zones. The sizes of the triangles will be almost entirely controlled by the number of points used in the walls (see, however, the *triangle\_area* keyword).
2. If a mesh composed of triangles or quadrilaterals already exists, the easier approach may be to write that information in the Sonnet format. Each mesh zone in the Sonnet file format is defined by five points. These coordinates are given on three consecutive lines. The “corners” appear on the first and third lines; the second line contains the “center” point. Inside *definegeometry2d*, the four corners must trace out the shape in a clockwise fashion. While there is some freedom in the way these points are labeled, for definiteness, we will say that the first line contains the second and third points (in that order) of a quadrilateral; the third line has the first and fourth points. For a triangle, just make the two points on the third line the same (the code will also agree to call it a triangle if the third and fourth points are the same). If the center point is set exactly to zero, built in routines will be used to compute an appropriate center. As in the above example, additional data about the hardware can be specified and translated into “walls”. The mesh edges can be used together with the walls to specify the rest of the polygons required to fill the universal cell.

The DEGAS 2 User’s Manual contains two example applications of *definegeometry2d*, one demonstrating the use of DG and the other illustrating the use of an external code to create walls and polygons out of flux surfaces.

## 5 Input File

Note that the name of the input file for `definegeometry2d` is not listed in `degas2.in`. Although this might be viewed as an omission, it is consistent in that `definegeometry2d` is, in principle, one of many different means of generating **DEGAS 2** geometries and, thus, is not as fundamental as the other entries in `degas2.in`.

Instead, the command line for the execution of `definegeometry2d` specifies the name of the main `definegeometry2d` input file:

```
definegeometry2d an_input_file
```

The input file, called here `an_input_file`, will contain pointers to other files that provide externally generated, detailed information about the geometry. Like other text input files used in **DEGAS 2**, blank lines, spaces, and lines beginning with a comment character # are ignored (comments can also appear at the end of a line).

The input file consists of two sections:

1. A preparatory stage in which global parameters are set and file names are specified,
2. A “construction” section containing the specification of polygons that will fill the universal cell.

Let us introduce the components of the input file with a nontrivial example:

```
# There are two parts to the input file. The first provides global
# information.
#
# It's a good idea to set the symmetry of the problem first.

symmetry cylindrical

# The bulk of the data describing the "hardware" in this problem are
# in a file generated by the DG code.

dg_file Run35/bypass_12.dgo

# The bounds keyword specifies the size of the universal cell.

bounds 0.3 1.1 -0.8 0.7

# Another big chunk of data required in setting up the problem is the
# mesh which will store the plasma-related data. In this case, the
# mesh is provided in the Sonnet format.

sonnet_mesh Run10/cmod.carre.008 61 26

# The end_prep command terminates the first part of the input file. The
# primary task undertaken with the execution of this command is the
# generation of plasma zones from the quadrilaterals described by the
# plasma mesh.

end_prep

# In the second part of the input file, the user must define zones that
# will completely fill (without overlap) the universal cell. These
# zones are specified as the union of one or more polygons. The
# new_zone command begins the definition of the next zone; in this
# case, a solid zone. The zone definition is terminated by the next
# "new_zone" command (or an "end" or "quit"). The code will assume that
# the polygon will be closed by connecting the last point specified
# to the first point. All polygons should be convex.

new_zone solid

# The following five polygons all belong to that single zone.
# The polygons are enumerated and labeled to facilitate external reference.

# The definition of a polygon is initiated by a "new_polygon" command
# and finished by "breakup_polygon", "triangulate_polygon", or
# "triangulate_to_zones". Each of these breaks the polygon down into
# DEGAS 2's internal description consisting of cells and surfaces.
# In the last case, each of the resulting triangles is assigned a new
# zone number.

#
```

```
# Keywords like "outer", "wall", and "edge" specify single points or a
# range of points used to construct the polygon. Other keywords like
# "stratum" or "material" specify properties associated with the polygon.
# Note, however, that these properties will apply to all subsequent
# polygons until overridden by new property settings. Finally, the
# "print_polygon" command (commented out since this input file was well
# past the testing stage) writes out the points in the polygon to a
# text file so they can be analyzed or plotted externally. This is
# useful for debugging problem polygons.

# Polygon 1: inner limiter
new_polygon
  stratum 1
  material mo      # For all solid zones (default temp. and recycling)
  outer 0 1
  wall 6 0 1
  wall 3 3 3
  edge 0 0 * reverse
#  print_polygon poly_1
breakup_polygon

# Polygon 2: upper divertor
new_polygon
  stratum 2
  outer 1 2
  wall 5 11 0
#  print_polygon poly_2
breakup_polygon

# Polygon 3: outer limiter
new_polygon
  stratum 3
  outer 2 3
  wall 5 49 11
#  print_polygon poly_3
triangulate_polygon

# Polygon 4: gas box, right side
new_polygon
  stratum 4
  outer 3 0
  wall 5 68 49
#  print_polygon poly_4
breakup_polygon

# Polygon 5: gas box, left side
new_polygon
  stratum 5
  outer 0
  edge 0 0 0 0
  wall 2 0 2
  wall 4 1 15
```

```
    wall 5 71 68
#      print_polygon poly_5
breakup_polygon

new_zone solid

# Polygon 6: outer target
new_polygon
stratum 6
wall 7 0 15
edge 61 61 *
wall 7 24 26
#      print_polygon poly_6
breakup_polygon

new_zone vacuum

# Polygon 7: private flux region
new_polygon
stratum 7
edge 0 10 0 0
edge 52 61 0 0
wall 4 6 0
wall 2 1 0
#      print_polygon poly_7
triangulate_to_zones

new_zone vacuum

# Polygon 8: gas box entrance
new_polygon
stratum 8
edge 61 61 0 0
#
# The following define two auxiliary sectors belonging to stratum 14
#
aux_stratum 14
wall 7 15 15
end_aux_stratum
wall 4 7 7
aux_stratum 14
wall 4 6 6
end_aux_stratum
#      print_polygon poly_8
breakup_polygon

new_zone vacuum

# Polygon 9: gas box
new_polygon
stratum 9
wall 7 15 3
```

```
    wall 5 51 71
    wall 4 15 7
#      print_polygon poly_9
    triangulate_to_zones

new_zone vacuum

# Polygon 10: gas box exit
    new_polygon
        stratum 10
        wall 7 3 3
#
# Here we add two more sectors to auxiliary stratum 14
#
        aux_stratum 14
        wall 7 2 2
        end_aux_stratum
        wall 5 50 50
        aux_stratum 14
        wall 5 51 51
        end_aux_stratum
#      print_polygon poly_10
    breakup_polygon

new_zone vacuum

# Polygon 11: lower main chamber
    new_polygon
        stratum 11
        edge 61 42 26 26
        wall 5 38 50
        wall 7 2 0
        wall 7 26 24
#      print_polygon poly_11
    triangulate_to_zones

new_zone vacuum

# Polygon 12: upper main chamber
    new_polygon
        stratum 12
        edge 42 0 26 26
        wall 3 3 3
        wall 6 1 1
        wall 5 0 38
#      print_polygon poly_12
    triangulate_to_zones

new_zone exit

# Polygon 13: core plasma
    new_polygon
```

```
stratum 13
edge 11 51 0 0
#      print_polygon poly_13
breakup_polygon

# Now define a diagnostic based on the auxiliary stratum defined above
# in polygons 8 and 10:

new_diagnostic Throat sectors
    stratum 14
end_diagnostic

# The description of this geometry in terms of polygons is written out
# to a netCDF file with this name so that it may be used again later.

polygon_nc_file Run35/bypass_12_poly_a.nc

# The "end" command instructs the code to finish setting up the
# internal geometry arrays, to check the geometry for consistency,
# and to write its netCDF file. Since the process of generating
# a geometry as complex as this one is understandably iterative, the
# code also has a "quit" command (that can be used anywhere except
# within a polygon definition) that quietly terminates the code without
# performing any of these finalization steps.

end
```

## 6 Keyword Reference

Each line in the input file is of the form: *keyword arguments* with some keywords having multiple arguments, others having none at all. Lower case single letters (perhaps with subscripts) are used to represent integer arguments. Upper case single letters correspond to real (i.e., floating point) arguments. All other arguments are strings. All keywords and strings should be lower case. This first group of keywords is intended only for use in the preparatory stage of the code; invoking them after the *end\_prep* command will likely result in the code crashing or behaving strangely.

## Preparatory Section

*symmetry* `sym` describes the symmetry of the geometry with `sym = plane, cylindrical` (“toroidal symmetry”), `oned` (for a planar system with two symmetry directions), `plane_hw` (nearly symmetric planar system with resolution in the *y* direction), `cylindrical_hw` (nearly cylindrically symmetric system with resolution in the toroidal direction), `cylindrical_section` (same as `cylindrical_hw`, but over a limited range of toroidal angles). The `symmetry` keyword must appear prior to the `end_prep` keyword.

`dg_file filename` specifies the name of an output file from the DG code (with the extension `.dgo`). If `filename` does in fact contain `.dgo`, the dimensions therein are assumed to be in millimeters; filenames without this extension are assumed to have data in meters (data perhaps generated by some other means). The coordinates of the “nodes” (or points) specified in the file are read in. Their ordering in the file is that of the “elements” (or segments) to which the nodes belong. A list of elements in terms of the node numbers is compiled as the file is read in. Each of the nodes is then characterized as being of a particular type:

1. `node_no_elements` if the node has no corresponding elements,
2. `node_one_element` if the node is associated with only one element,
3. `node_regular` if the node is associated with two elements with normals of the same sense,
4. `node_mixed_normals` if the node is associated with two elements having normals in opposite sense (i.e., the node is either the “start” or the “end” of both elements),
5. `node_many_elements` if the node represents the intersection of more than two elements.

Once each of the nodes has been characterized, the resulting information permits the translation of the elements into “walls” that can be used in the subsequent specification of polygons. The code loops over all of the irregular nodes (that are connected to at least one element). Each of these serves as the start of a wall. The rest of the wall is mapped out from the adjacent elements using the well-defined connectivity associated with the regular nodes comprising those elements. The process is stopped, of course, when the trail ends at another irregular node. A subsequent search of the elements that have not been assigned to walls is performed to identify closed surfaces consisting of only regular nodes. The upshot of this procedure is to be cognizant of the location of irregular nodes when using DG. Handling them carefully there can result in a simpler set of walls and, hence, ease the task of designing the polygons.

Suitably updated versions of DG permit the polygons to be defined there first using DG’s graphical interface. DG counts these as they are defined; the resulting integer label is used with the `dg_polygon` keyword in the second part of the `definegeometry2d` file. In the DG output file, the polygons are delimited by a line of the form “`polygon i`”, where “*i*” is this integer label. Inside that polygon specification is the “wall” section, a list of the DG elements comprising that polygon. This is followed by a “polymat” parameter that is currently not being used; it may eventually be used to set the material associated with a polygon. Finally, the sections “`meshcon1`” and “`meshcon2`” describe connections between the elements in the polygon’s “wall” and the outer edge of the plasma mesh associated with the DG file. The code that processes the DG file assumes that these sections of the polygon specification appear in precisely this order.

`sonnet_mesh filename  $n_x$   $n_z$   $i_{x,min}$   $i_{x,max}$   $i_{z,min}$   $i_{z,max}$`  specifies a file, `filename`, containing a plasma mesh in the Sonnet format. Files not generated by Sonnet (or, equivalently, by Carre) can be used, but the user should familiarize themselves with the details of the routine that reads the file, `read_sonnet_mesh`. The Sonnet format specifies the plasma mesh as a  $n_x$  by  $n_z$  rectangular grid of quadrilaterals. The designation of the  $x$  and  $z$  directions is somewhat arbitrary and does not necessarily have any correspondence with the (Cartesian)  $X$  and  $Z$  coordinates referred to elsewhere in this program. The usual tokamak configuration has the  $x$  coordinate following along flux surfaces and the  $z$  coordinate going across flux surfaces. Each quadrilateral is given as a set of four corners and a center so that the mesh can be treated as completely unstructured and no connectivity information is needed. On the other hand, corners of adjacent mesh cells that are intended to match should match *exactly*. In defining the surrounding polygons, the user will undoubtedly need to be familiar with the mesh's connectivity in defining the surrounding polygons. The ordering of the corners is important. See below and in subroutine `read_sonnet_mesh` for more details.

If the entire mesh is to be read in, only  $n_x$  and  $n_z$  parameters need to be specified. The other four parameters allow the user to describe and use a rectangular subset of the mesh. One might want to do this if the mesh is physically much larger than the volume of interest. By isolating a subset of the mesh with these four parameters, its boundaries can be used to construct polygons via the `edge` keyword in the second part of this input file. Once the mesh subset has been read in, the original mesh indexing is forgotten. The code will proceed as if an entire mesh of size  $n_x = i_{x,max} - i_{x,min} + 1$  by  $n_z = i_{z,max} - i_{z,min} + 1$  were read in. The user will need to be mindful of this index shifting when invoking the `edge` keyword.

Note that these parameters are used as if the mesh indices begin at 1. Since the Sonnet format typically starts the mesh indices at 0, the values of  $n_x$  and  $n_z$  should be 1 greater than the indices of the last cell in the Sonnet file. The same is true for the subset indices so that, for example,  $i_{x,min} \geq 1$  and  $i_{x,max} \leq n_x$ .

`uedge_mesh filename` specifies the name of a file, `filename`, generated by the UEDGE code for the purpose of transferring data to DEGAS 2. The file contains both geometry and plasma data, although only the former is read in by the `read_uedge_mesh` routine. The first few lines of the file contain additional information describing the mesh. The first line is assumed to be a comment. The first number on the second line is assumed to be  $n_x$ . Likewise, the third line provides  $n_z$ . The next three lines are ignored. As with the Sonnet format, a center and four corners are given for each quadrilateral and no connectivity information is needed. See the routine `read_uedge_mesh` for more details on the order of the corners.

`wallfile filename with_sonnet` specifies the name of a file, `filename`, that contains a list of walls, each comprised by two or more points. Like other DEGAS 2 text files, `wallfiles` can contain blank lines and comments (beginning with #). Spacing should not matter. The first line provides the number of walls in the file, `num_new_walls`. The following line or lines list the number of points comprising each wall. I.e., these lines must contain `num_new_walls` integers. The rest of the file contains the coordinates in ordered pairs, with one ( $X, Z$ ) pair per line, starting with the first point of the first wall. The other points in that wall follow on consecutive lines. The points in the second wall (if there is one) come after that. Comments can be inserted between walls to facilitate legibility. For simple one wall systems, the file can be easily created by hand. For more complicated systems involving several walls, the user may benefit from writing a code to generate the file.

The optional argument `with_sonnet` should be used if the coordinates in the `wallfile` are supposed to coincide with coordinates in a Sonnet format mesh. Since the latter are restricted to 10 digits of precision, a similar truncation is applied to the `wallfile` coordinates in this case.

`print_min_max` loops over all of the nodes (read in via the `dg_file` or `wallfile` keywords) and prints to standard out the maximum and minimum  $R$  and  $Z$  values (in meters). These can be useful in choosing suitable bounds for the universal cell.

*bounds*  $X_{\min}$   $X_{\max}$   $Z_{\min}$   $Z_{\max}$   $Y_{\min}$   $Y_{\max}$  specifies the corners of the universal cell in  $X$  ( $X_{\min} \rightarrow X_{\max}$ ),  $Z$  ( $Z_{\min} \rightarrow Z_{\max}$ ), and in nearly symmetric 3-D cases  $Y$  or  $\phi$  ( $Y_{\min} \rightarrow Y_{\max}$ ). The *outer* keyword can be used to incorporate these points into polygons.

The  $Y$  values are not needed for 2-D (*symmetry = plane* or *cylindrical*) or full torus 3-D (*symmetry = cylindrical\_hw*) cases. For nearly symmetric 3-D plane cases (*symmetry = plane\_hw*), the  $Y$  values are in meters. For 3-D cylindrical section cases (*symmetry = cylindrical\_section*), the  $Y$  values describe the range of toroidal angles in degrees that are to be used in the problem (note, however, that these are converted to radians during processing). Negative values are permitted, e.g., so that the user can put  $\phi = 0$  in a convenient location. The only constraint is that  $0 < Y_{\max} - Y_{\min} < 360^\circ$ . When the universal cell is defined with the invocation of the *end\_prep* command, it will have surfaces at  $\phi = Y_{\min}$  and  $\phi = Y_{\max}$ . For the full torus 3-D (*cylindrical\_hw*) case,  $Y_{\min}$  and  $Y_{\max}$  may be specified; if they are not, they are assumed to be  $Y_{\min} = 0$  and  $Y_{\max} = 360^\circ$ .

*print\_walls format filename* instructs the code to write out the current set of walls. If *filename* is not provided, the data are written to standard out. There are two possible *formats*. The first, *tabular*, has the walls laid out in successive columns, suitable for plotting with an external program. The second, *linear*) is consistent with the format used by the *wallfile* keyword.

*uniform\_ys ny* sets up a uniform grid in the  $y$  or  $\phi$  direction of nearly symmetric 3-D problems. This keyword can only be used with *symmetry* values *plane\_hw*, *cylindrical\_hw*, and *cylindrical\_section*. In all three cases,  $n_y$  uniformly spaced  $y = \text{constant}$  or  $\phi = \text{constant}$  surfaces will be defined, breaking the problem up into equal size  $n_y - 1$  zones in the third dimension.

In the *plane\_hw* and *cylindrical\_section* cases, the first and last of these surfaces are offset from the universal cell surfaces at  $Y_{\min}$  and  $Y_{\max}$  by a small amount,  $0.01(Y_{\max} - Y_{\min})$ , so that border zones can be defined there.

*y-values filename OR y<sub>1</sub> y<sub>2</sub> y<sub>3</sub> ...* This keyword allows the user to directly specify the grid in the  $y$  or  $\phi$  direction for nearly symmetric 3-D problems. This keyword can only be used with *symmetry* values *plane\_hw*, *cylindrical\_hw*, and *cylindrical\_section*. In planar cases, the  $y_i$  values are in meters; in cylindrical cases, they are in degrees (but converted to radians internally). In all instances, the user must have  $Y_{\min} \leq y_i \leq Y_{\max}$  with the  $y_i$  monotonically increasing with  $0 \leq i \leq n_y - 1$  (there will be  $n_y$  surfaces defined at the specified  $y$  values; hence, there are  $n_y - 1$  segments in the third dimension). In the bounded cases (i.e., not *cylindrical\_hw*), a small gap is needed between the first and last surfaces and the bounding universal cell surfaces,  $y_0 > Y_{\min}$  and  $y_{n_y-1} < Y_{\max}$ . In full toroidal cases (*cylindrical\_hw*), the code expects  $y_0 = Y_{\min}$  and  $y_{n_y-1} = Y_{\max}$ . If the list of  $y$  values is too long to fit on a single line, they can instead be put into a file *filename*, one per line. This also facilitates commenting as in other DEGAS 2 text input files. Note that this keyword should appear after *symmetry* and *bounds* since the corresponding input information is used in checking the *y-values*.

*end\_prep* terminates the preparatory stage of the code. Dynamically allocated arrays associated with nodes, elements, and walls are trimmed to their final sizes, and the universal cell is established. If a plasma mesh has been specified via the *sonnet\_mesh* or *uedge\_mesh* keywords, the constituent quadrilaterals are translated into polygons and then decomposed into the internal elements of the DEGAS 2 geometry with each quadrilateral corresponding to a plasma zone. One advantage of making these the first zones defined in the problem is that transfer of zone-based output information to other codes (such as a fluid plasma code) is simplified.

**Construction Section** The rest of the keywords are associated with the “construction” section of the input file. While they may (even legally) be used prior to the *end\_prep* command, such usage is not encouraged in the interest of maintaining the simplicity of the input file. One exception is the *quit* keyword that can be used anywhere in the input file (except inside a polygon definition). There is a further subdivision in that all keywords regarding a specific polygon must appear between its *new\_polygon* command and the command that denotes its completion (*clear\_polygon*, *breakup\_polygon*, *triangulate\_polygon*, or *triangulate\_to\_zones*). Likewise, all of the keywords pertaining to a diagnostic must appear between its *new\_diagnostic* and *end\_diagnostic* commands.

*new\_zone type alt\_type i<sub>1</sub> i<sub>2</sub> i<sub>3</sub> ...* starts definition of a new zone. All polygons specified after this command and prior to the next *new\_zone* command will be added to this zone. For 2-D problems, just the first argument, *type*, is allowed. The possible values of *type* are *vacuum*, *plasma*, *solid*, and *exit*.

In nearly symmetric 3-D cases (*symmetry* must be *plane\_hw*, *cylindrical\_hw* or *cylindrical\_section*), the zone type can be varied in the third dimension. If no variation is desired, again specify just the *type* with the *new\_zone* command. To use mixed zone types, use the first argument *type* to specify the dominant zone type. The second zone type and its locations are determined by the remaining arguments. The *alt\_type* provides the zone type and again must be *vacuum*, *plasma*, or *solid*. The following indices correspond to the grid in the *y* or  $\phi$  locations specified by the *uniform\_ys* or *y\_values* keyword. The first segment of the grid corresponds to *i* = 0; the last to *i* = *n<sub>y</sub>* - 2 (since there are *n<sub>y</sub>* - 1 segments in the third dimension).

For convenience, a range of *i* values can be specified as *i - j* (with a space on either side of “-”). That is, the range “3 4 5 6” could be abbreviated as “3 - 6”.

Zones of mixed type must be decomposed with the *triangulate\_to\_zones* command. Note that currently this option cannot be used with exit zones. Nearly the same functionality can be obtained by specifying a solid zone with a recycling coefficient of zero.

*new\_polygon* begins the definition of a new polygon. The following keywords (down to and including *triangulate\_to\_zones*) can be used in describing this polygon.

*outer i j k* adds to the current polygon the *i*th, *j*th, and *k*th points of the bounding rectangle of the problem (defined with the *bounds* keyword, i.e., the minimum and maximum *X* and *Z* used to define DEGAS 2’s “universal cell”). The numbering of the corners is clockwise:

- 0 (or 4) (*X<sub>min</sub>*, *Z<sub>min</sub>*)
- 1 (*X<sub>min</sub>*, *Z<sub>max</sub>*)
- 2 (*X<sub>max</sub>*, *Z<sub>max</sub>*)
- 3 (*X<sub>max</sub>*, *Z<sub>min</sub>*)

There can be any number of arguments (e.g., see the above example input file).

*wall i<sub>wall</sub> j<sub>start</sub> j<sub>stop</sub> reverse* takes points from wall number *i<sub>wall</sub>* that was specified with either the *wallfile* or *dg\_file* command. The parameters *j<sub>start</sub>* and *j<sub>stop</sub>* prescribe which points from that wall to use. Note that the first point of the wall is numbered 0 so that the only valid values of *j<sub>start</sub>* and *j<sub>stop</sub>* are between 0 and the number of elements in the wall. If the wall needs to be traversed in the reverse direction, *j<sub>stop</sub>* can be set less than *j<sub>start</sub>*. If *j<sub>start</sub>* = *j<sub>stop</sub>*, a single point is added to the polygon. If both *j<sub>start</sub>* and *j<sub>stop</sub>* are replaced by a single \*, all of the points in the wall are used. The wildcard character can also be used as: *j<sub>start</sub> \** to indicate that all of the points from *j<sub>start</sub>* to the end of the wall should be used. The additional argument *reverse* can be added at the end of the line to reverse the ordering of the points indicated by *j<sub>start</sub>*, *j<sub>stop</sub>*, and / or \*. The primary usage of *reverse*, however, will be in conjunction with \*.

`edge  $i_{x,\text{start}}$   $i_{x,\text{stop}}$   $i_{z,\text{start}}$   $i_{z,\text{stop}}$   $\text{xcut}$   $\text{reverse}$`

takes an “edge” from a plasma mesh defined with either the `sonnet_mesh` or `uedge_mesh` keywords and adds it to the current polygon. To properly refer to an edge, the arguments must be such that either  $i_{x,\text{start}} = i_{x,\text{stop}}$  or  $i_{z,\text{start}} = i_{z,\text{stop}}$ . Valid values of  $i_{x,\text{start}}$  and  $i_{x,\text{stop}}$  will be between 0 and  $n_x$ , inclusive. Likewise,  $i_{z,\text{start}}$  and  $i_{z,\text{stop}}$  will be between 0 and  $n_z$ . Either pair can be replaced with  $*$  to indicate that the entire edge should be added to the polygon. The code used to process the arguments “knows” about the four corners comprising each mesh zone so that `edge * 0 0` actually refers  $n_x + 1$  points (see also the additional detail below on corners). If a subset of a Sonnet mesh is being used, these values are relative to it, not to the original, full mesh.

The one instance where this is not quite true is at a discontinuity (a “cut”) in the  $x$  direction of the computational mesh. On the “right” side of the cut (“right” in the topological sense; i.e., the mesh cells  $i_x$  and  $i_x - 1$  do *not* share a common edge), adding the optional `xcut` argument to the `edge` command with  $i_{x,\text{start}} = i_{x,\text{stop}} = i_x$  will allow the edge of the  $i_x$  cell along this cut to be added to the polygon. Without the `xcut` argument, this command would instead yield segments following the boundary between the  $i_x$  and  $i_x + 1$  cells. On the “left” side of the cut (the mesh cells  $i_x$  and  $i_x + 1$  do not share a common edge), the usual `edge` command with  $i_{x,\text{start}} = i_{x,\text{stop}} = i_x$  will work as is, and the `xcut` command is not needed. The `xcut` argument may also be needed in conjunction with an  $i_{y,\text{start}} = i_{y,\text{stop}}$  edge to specify a single point on the “right” side of a cut. Doing so requires a *separate* `edge` command for that point (i.e., it would have both  $i_{x,\text{start}} = i_{x,\text{stop}}$  and  $i_{y,\text{start}} = i_{y,\text{stop}}$ , as well as the `xcut` argument). The specific implementation of the `xcut` option is described below under “Other Considerations”.

As with the `wall` keyword, the `reverse` argument can be used to indicate that the points should be added to the polygon in the order opposite to that indicated by the other arguments. One, none, or both of the `xcut` and `reverse` options may appear. If both are present, they do not need to be in a particular order.

`dg_polygon  $i_{\text{polygon}}$`  adds the segments associated with the  $i_{\text{polygon}}$ th polygon defined in DG to the current polygon. This functionality allows the user to set up polygons in DG rather than going through the more tedious and error prone process of defining them in this input file with the `wall` and `outer` keywords. Different types of DG polygons are possible:

1. If the DG polygon defines a closed figure, no other points need be specified for this `new_polygon` group, although the user should still assign `stratum`, `material`, `temperature`, and `recyc_coef` values as needed. The resulting polygon will have a clockwise orientation.
2. If the polygon is not closed, it can be closed by various means,
  - (a) The DG polygon’s “mesh connection” data identify a single point on the edge of the plasma mesh. This point will be used to close the polygon without any additional input from the user.
  - (b) The DG polygon’s “mesh connection” data identify a pair of points on edge of the plasma mesh, with the implication that the set of mesh edge elements “between” them will be used to close the polygon. However, this set is uncertain since it could consist of either the clockwise or counter-clockwise path between the specified points; the user must invoke an `iedge` command to determine which should be used.
  - (c) No “mesh connection” data are specified. This would be the case if the user wishes to complete the polygon with another command (e.g., `edge` or `wall`); `dg_polygon` must be the first command to add points to the polygon in this case. This approach is not recommended since the orientation of the DG polygon may not be as expected. If that is the case, the user can invoke the `reverse` command immediately after `dg_polygon`.

`iedge edge1 ...` specifies one or more edges of plasma mesh defined with either the `sonnet_mesh` or `uedge_mesh` keywords. The edges are identified by the strings *S*, *E*, *N*, and *W* (referring to south, east, north, and west), corresponding to the  $i_z = 0$ ,  $i_x = n_x$ ,  $i_z = n_z$ ,  $i_x = 0$  boundaries, respectively.

*reverse* instructs the code to reverse the order of all points that have been added to the current polygon; the polygon need not be complete.

*stratum*  $i_{\text{stratum}}$  labels the polygon as belonging to stratum number  $i_{\text{stratum}}$ . It will be applied to subsequent polygons until used again.

*material*  $\mathtt{mat}$  sets the material for the polygon to the material having the symbol  $\mathtt{mat}$ . A material with this symbol must appear in the *materials\_infile* (named in the *degas2.in* file) for this run. It will be applied to all subsequent polygons until used again. Note that this setting only affects the calculation when one or more edges of this polygon ends up being a target or wall sector.

*temperature*  $T$  associates a temperature of  $T$  Kelvin (a real number) to the polygon. Its value currently is used only to determine the energy of desorbed species, although other applications may be incorporated. It will be applied to all subsequent polygons until used again. Note that this setting only affects the calculation when one or more edges of this polygon ends up being a target or wall sector.

*recyc\_coef*  $R$  associates a recycling coefficient of  $R$  ( $R$  a real number,  $0 \leq R \leq 1$ ) to the polygon. It will be applied to all subsequent polygons until used again. Note that this setting only affects the calculation when one or more edges of this polygon ends up being a target or wall sector. Setting  $R = 0$  allows a purely absorbing boundary to be modeled.

*triangle\_area*  $A$  specifies a minimum area in square meters for the triangles to be created by **Triangle** (with  $A$  a real number) when the *triangulate\_to\_zones* command is executed. Practically, it seems that this ends up as little more than a suggestion. The reason is that we cannot allow **Triangle** to add points along the boundary of the polygon. Although this would not compromise the definition of DEGAS 2 surfaces from the triangulation of the current polygon, the surface(s) of the adjacent polygon(s) would not be subdivided in the same way. Hence, the exact correspondence of adjacent surfaces that the code relies upon for establishing connectivity would be destroyed. **Triangle** can, however, add points to the interior of a polygon; this parameter  $A$  can be used to control that process.

*triangle\_hole*  $n X Z$  specifies an integer number of holes,  $n$ . This is currently constrained to be a single hole,  $n = 1$ . The coordinates (real numbers) of the hole are given as  $X$  and  $Z$ . The hole location is used by **Triangle** when either the *triangulate\_polygon* or *triangulate\_to\_zones* command is executed on a polygon that encircles a region that is *not* to be broken up into triangles. The hole location can be anywhere in this interior region.

*aux\_stratum*  $i_{\text{stratum}}$  initiates the definition of an auxiliary stratum (consisting of one or more auxiliary sectors) labeled as stratum number  $i_{\text{stratum}}$ . The definition is terminated by an *end\_aux\_stratum* command. All of the polygon segments specified between the two commands will be used to define auxiliary sectors in this stratum. More precisely, sectors will be set up starting at each *point* between the two commands; the end point of each sector being the next point in the polygon. Thus, the end point of the last auxiliary sector is the first point specified *after* the *end\_aux\_stratum* command. For this to work correctly, *the polygon must have a clockwise orientation*. The *aux\_stratum* and *end\_aux\_stratum* commands may be invoked multiple times across various polygons, with the same or different values of  $i_{\text{stratum}}$ . In this way, segments of multiple polygons can be incorporated into a single stratum.

*end\_aux\_stratum* terminates the definition of an auxiliary stratum begun with an *aux\_stratum* command.

*print\_polygon* *filename* prints the points of the current polygon to a file, *filename*. If *filename* is not present, the coordinates are written to standard out.

*clear\_polygon* will cause the code to completely forget about this polygon in subsequent operations. This is useful for checking the integrity of several polygons with a single run of the code.

*breakup\_polygon* instructs the code to use DEGAS 2's original *decompose\_polygon* routine to break the polygon up into the internal geometry elements. This routine expects the polygon to be traced out in a clockwise direction. Note also that *decompose\_polygon* is far from foolproof and will occasionally fail with a "unable to decompose polygon" error. This problem can usually be remedied by choosing a different point of the polygon as the first point. On the other hand, the functionality of the vastly more robust *triangulate\_polygon* routine is equivalent. So, try using it instead when this error arises. The two approaches may result in different run efficiencies, but no such variability has ever been established.

*triangulate\_polygon* instructs the code to use the **Triangle** package to break the polygon up into DEGAS 2's internal geometry elements. The **Triangle** routine is extremely tolerant and robust; the input polygon can be traced out in either direction. However, a clockwise orientation is recommended since this will preserve (as *strata\_segment* labels) the relationship between the resulting bounding segments and the original polygon points. This command does only a basic triangulation and is only recommended for use with polygons in solid zones. In nearly symmetric 3-D cases, these zones *will not* be resolved in the  $i_y$  direction.

*triangulate\_to\_zones* instructs the code to use the **Triangle** package to break the polygon up into DEGAS 2's internal geometry elements. The **Triangle** routine is extremely tolerant and robust; the input polygon can be traced out in either direction. Each resulting triangle is added to the geometry as a separate zone (of the current type). A second refining call to **Triangle** beyond the one analogous to that done with *triangulate\_polygon* is performed that permits the application of additional constraints such as a minimum area. A clockwise orientation is recommended since this will preserve (as *strata\_segment* labels) the relationship between the resulting bounding segments and the original polygon points. This option also ensures that zones are resolved in the  $i_y$  direction in nearly symmetric 3-D problems. **Note: the next polygon to appear in the file must be preceded by a *new\_zone* command.** (This is the last of the polygon-specific keywords.)

*new\_diagnostic diagnostic name* initiates the definition of a diagnostic group (of sectors) associated with the name *diagnostic name* (the name of the diagnostic may contain spaces). **The following keywords can be used to describe this diagnostic.** An *end\_diagnostic* command terminates its definition. The only other command that must be used is *stratum*; the others listed below are optional.

*stratum i<sub>stratum</sub> type* designates a previously defined stratum that will determine the sectors to be used for this diagnostic. If *i<sub>stratum</sub>* is associated with a default sector (i.e., corresponds to a stratum number specified with the *stratum* keyword inside a polygon or diagnostic definition), the optional argument *type* determines which of the two types of default sectors will be used. If *type* is set to **solid**, the diagnostic will be comprised of the wall, target, or exit sectors created from that polygon with that stratum label. If the *type* argument is omitted, its value is assumed to be **solid**. Alternatively, if *type* is set to **nonsolid**, the adjoining plasma or vacuum sectors are used. When an auxiliary stratum is used, all sectors of that stratum will be incorporated.

This capability can be used to duplicate and / or extend (e.g., to increase resolution in energy or angle of incidence) the default diagnostics and associated tallies without having to modify the code.

*variable var* requests that an independent variable mesh for the physical quantity *var* be set up for the diagnostic being defined by a preceding *new\_diagnostic* command. The default is no variable; in that case, the diagnostic lumps together all flights passing through its sectors. By using **energy** or **angle** for *var*, the user can resolve these passages according to the flights energy or angle (angle of incidence relative to the normal to the sector), respectively. The user can control the units used for the energy mesh by instead using for *var* **energy J** or **energy eV** (the default is eV). Likewise, the angle can be in degrees, **angle degrees** (default) or radians, **angle radians**.

*number n<sub>var</sub>* specifies the number of values in the independent variable mesh for the diagnostic being defined in conjunction with a preceding *new\_diagnostic* command.

*minimum*  $V_{\min}$  specifies the minimum value of an independent variable mesh being defined in conjunction with a preceding *new\_diagnostic* command.

*maximum*  $V_{\max}$  specifies the maximum value of an independent variable mesh being defined in conjunction with a preceding *new\_diagnostic* command.

*multiplier*  $F$  specifies a multiplier to be applied to the *maximum* and *minimum* of a diagnostic independent variable mesh. This command allows the user to utilize units *other than* those described above under *variable*. The multiplier must result in MKS units for the mesh (i.e., J for energy and radians for angle).

*spacing* **spac** specifies the spacing of an independent variable mesh being defined by a preceding *new\_diagnostic* command. The only valid values for **spac** are **linear** and **log**. No default is set, so one of these must appear explicitly.

*end\_diagnostic* terminates the definition of the diagnostic begun with a preceding *new\_diagnostic* command.

**y\_min\_zone mat**  $i_{\text{stratum}}$   $i_{p,\text{stratum}}$  specifies that the  $Y_{\min}$  boundary of bounded, nearly symmetric 3-D cases (*symmetry = plane-hw* or *cylindrical\_section*) is to be assigned the material with symbol **mat**. A material with this symbol must appear in the *materials\_infile* (named in the *degas2.in* file). Sectors based on this zone will be labeled with stratum number  $i_{\text{stratum}}$ . If the optional second stratum label  $i_{p,\text{stratum}}$  is present, sectors based on the adjacent (corresponding to  $i_y = 0$ ) zone are also defined and assigned stratum label  $i_{p,\text{stratum}}$ .

**y\_max\_zone mat**  $i_{\text{stratum}}$   $i_{p,\text{stratum}}$  specifies that the  $Y_{\max}$  boundary of bounded, nearly symmetric 3-D cases (*symmetry = plane-hw* or *cylindrical\_section*) is to be assigned the material with symbol **mat**. A material with this symbol must appear in the *materials\_infile* (named in the *degas2.in* file). Sectors based on this zone will be labeled with stratum number  $i_{\text{stratum}}$ . If the optional second stratum label  $i_{p,\text{stratum}}$  is present, sectors based on the adjacent (corresponding to  $i_y = n_y - 2$ ) zone are also defined and assigned stratum label  $i_{p,\text{stratum}}$ .

**polygon\_nc\_file** **filename** will cause the netCDF file containing the polygon description of the geometry (based on the g2 class, as described in the file **geometry2d.hweb**) to be named **filename**. The default file name is **polygon.nc**.

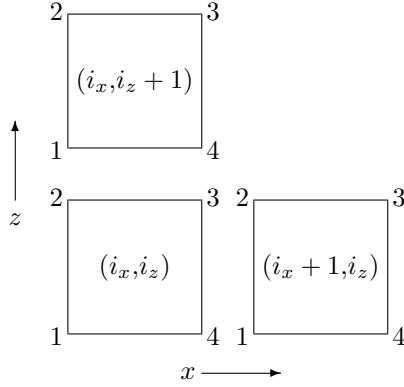
**quit** causes the code to terminate immediately. This is useful for preliminary runs in which the geometry is known to be inconsistent or incomplete, but the code needs to be run so that walls or polygons can be printed out for external examination.

**end** denotes completion of the geometry specification. A variety of derived relationships are set up by subroutine calls. The geometry is checked (thoroughly) and written to its netCDF file.

## 7 Other Considerations

*Regarding corners with the edge keyword:* Because the **UEDGE** and **Sonnet** mesh formats both provide four corners for each mesh zone, the number of points along each “edge” of the mesh is one greater than the number of zones along that edge. This fact must be taken into account in defining polygons along these edges.

For clarity, here is the convention used to number the corners in *definegeometry2d*:



In the **Sonnet** mesh format, the first line contains corners 2 and 3. The second line specifies the center. Corners 1 and 4 are on the third line. The **UEDGE** mesh format specifies the corners for each point in the order 0 (center), 1, 4, 2, and 3.

For a particular  $(i_x, i_z)$  specified via arguments to the *edge* keyword, the logic that determines the corner to be used is:

1. If  $i_x = 0$  and  $i_z = 0$ , corner 1 of the  $(1, 1)$  zone (recall that there is no  $(0, 0)$  zone).
2. If  $i_x = 0$ , but  $i_z \neq 0$ , corner 2 of the  $(1, i_z)$  zone.
3. If  $i_x \neq 0$ , but  $i_z = 0$ , corner 4 of the  $(i_x, 1)$  zone.
4. Otherwise, use corner 3 of the  $(i_x, i_z)$  zone.

If the *xcut* argument for the *edge* keyword has been invoked, a separate set of logic is required:

1.  $i_x \neq 0$  is asserted. The above logic already handles this case.
2. If  $i_z = 0$ , corner 1 of the  $(i_x, 1)$  zone.
3. Otherwise, use corner 2 of the  $(i_x, i_z)$  zone.

In this way, the “left” edge of the  $i_x$  zone can be specified; the usual logic results in the “right” edge (except at  $i_x = 0$ ).

*Regarding the definition of detectors:* Ideally the user would also be able to specify detectors via this input file. Instead, a placeholder subroutine called *detector\_setup* has been defined in the file *def2ddetector.web*. The user can write a replacement routine that will be loaded in automatically if it is in a file named *usr2ddetector.web*. An example is contained in the file *btopdetector.web*. In practice, one would use such a file by executing commands analogous to:

```
cd $HOME/degas2/src
cp btopdetector.web usr2ddetector.web
touch usr2ddetector.web
cd ../SUN
gmake definegeometry2d
```

Likewise, to revert to the default (null) subroutine,

```
cd $HOME/degas2/src
rm usr2ddetector.web
touch def2ddetector.web
cd ../SUN
gmake definegeometry2d
```

Note the use of the **touch** command to make sure **gmake** recognizes the change.

*Regarding the orientation of sectors:* Sectors are defined using a zone and a surface bounding that zone. A flight leaving that zone through that surface is considered to be traveling “out” of that sector, establishing the applicability of the direction-sensitive variables available in **tallysetup**. For example, the flight would make a contribution to a sector-based tally for which the dependent variable is **mass\_out**, but not for one in which the variable is **mass\_in**.

The default diagnostics (see the “Diagnostic Sectors” section of the User’s Manual) are comprised of default wall, target, and exit sectors. So, flights traveling out of adjacent plasma or vacuum zones “into” a sector incorporated into one of these diagnostic groups will make contributions to corresponding tallies that have a dependent variable of **mass\_in**, **momentum\_in\_vector**, or **energy\_in**; these tallies represent sinks for the test species.

If the plasma-material interaction processes that result from the flight striking this surface yield one or more products to be tracked back into the plasma or vacuum, contributions will be made to tallies based on the same diagnostic group but having an outwardly directed dependent variable (e.g., **mass\_out**).

The “auxiliary sectors” capability provided by this code allows the definition of:

1. Wall, target, or exit sectors at locations of specific interest. Such sectors will work just as the corresponding default sectors do.
2. Vacuum or plasma sectors at locations of specific interest. In nearly symmetric 3-D cases, these might be used to determine the variation of fluxes in the third dimension. These can also be used to track fluxes between adjoining plasma or vacuum zones, where there are no default sectors at all.

The former are set up using the *aux\_stratum* and *end\_aux\_stratum* commands inside a solid polygon definition clause. The latter would have these keywords inside a vacuum or plasma polygon definition. The direction-sensitive variables for the former behave as above; the latter work in the opposing manner. I.e., a **mass\_in** based tally represents a source for a plasma or vacuum sector. Since nothing should be lost across the interface, the “**mass\_in**” contribution to a solid sector should equal the **mass\_out** of an adjoining plasma or vacuum sector (of the same extent in the third dimension), and vice versa.

*Special note on comments and debuggers:* Part of this file is written in C. The current Sun Workshop debugger (there may be others) utilizes the original FWEB file in stepping through the source code of the C routines. Getting the object file and the source code lines to match up requires exercising some care in writing comments. First, do not use the // comment style on a single line. Second, do not put blank lines before or after multiline (delimited by /\* \*/ ) comments. There may be other requirements as well.

```
$Id: 5e44bf6b93ddf62732935db1a92b3507fe550512 $

"definegeometry2d.f" 1≡
@m FILE 'definegeometry2d.web'

@m dg_loop #:0
@m meshcon_loop #:0
@m wall_loop #:0
@m wall_loop2 #:0
@m wall_break #:0
@m sector_break #:0
@m sector_break2 #:0
@m next_surface #:0
@m y_loop #:0
@m y_loop2 #:0
@m mixed_type_loop #:0
```

The unnamed module.

```
"definegeometry2d.f" 7.1≡
@Lc:   ⟨ Functions and subroutines 8 ⟩
#define REAL double // Need to be sure this does not overlap with local macros
#include <stdio.h>
#include "triangle.h" // Dependency explicitly included in Makefile
⟨ C Functions 36 ⟩C
```

## 8 The Main Program

```
"definegeometry2d.f" 8 ≡
@m element_start 0
@m element_end 1

@m node_UNDEFINED 0
@m node_regular 1
@m node_no_elements 2
@m node_one_element 3
@m node_many_elements 4
@m node_mixed_normals 5

@m sec_UNDEF 0
@m sec_p1 1
@m sec_p2 2
@m sec_miss 3
@m sec_skip 4
@m sec_dg_poly 5
@m sec_dg_wall 6
@m sec_dg_meshcon1 7
@m sec_dg_meshcon2 8
@m sec_done 40 // Try to match to maximum?

@m meshcon_max 2 // Number of meshcon sections in DG file
@m meshcon_elem_max 2 // Number of elements in a meshcon
@m mesh_h 0 // Labels for elements
@m mesh_v 1

@m poly_stratum 1 // Indices for poly_int_props
@m poly_material 2
@m poly_num_holes 3
@m poly_int_max 3

@m poly_temperature 1 // Indices for poly_real_props
@m poly_recyc_coef 2
@m poly_min_area 3
@m poly_hole_x 4
@m poly_hole_z 5
@m poly_real_max 5

@m clear_polygon 0
@m breakup_polygon 1
@m triangulate_polygon 2
@m triangulate_to_zones 3

@m max_triangles 5500
@m max_grp_sectors 8000

@m increment_num_elements num_elements++
dim_elements = max(dim_elements, num_elements)
var_realloc(dg_elements_list)
var_realloc(element_missing)
var_realloc(element_skipped)
var_realloc(element_assigned)
```

```

@m increment_num_nodes num_nodes++
dim_nodes = max(dim_nodes, num_nodes)
var_realloc(nodes)
var_realloc(node_type)
var_realloc(node_element_count)

@m increment_num_dg_poly num_dg_poly++
dim_dg_poly = max(dim_dg_poly, num_dg_poly)
var_realloc(dg_polygons)
var_realloc(dg_poly_num_elements)
var_realloc(dg_poly_num_meshcon)
var_realloc(dg_poly_meshcon)
var_realloc(dg_poly_meshcon_hv)

@m increment_num_walls num_walls++
dim_walls = max(dim_walls, num_walls)
var_realloc(wall_nodes)
var_realloc(wall_elements)
var_realloc(wall_segment_count)

@m increment_y_div y_div++
dim_y = max(dim_y, y_div)
dim_ym = max(dim_ym, y_div)
var_realloc(y_values)
var_realloc(facearray)
var_realloc(zonearray)
var_realloc(zone_type_array)

@m increment_g2_num_polygons g2_num_polygons++
var_realloc(g2_polygon_xz)
var_realloc(g2_polygon_segment)
var_realloc(g2_polygon_points)
var_realloc(g2_polygon_zone)
var_realloc(g2_polygon_stratum)
var_realloc(poly_int_props)
var_realloc(poly_real_props)
g2_polygon_stratumg2_num_polygons = int_unused
do i_inc_g2 = 0, g2.num_points - 1
  g2_polygon_xzg2_num_polygons, i_inc_g2, g2_x = zero
  g2_polygon_xzg2_num_polygons, i_inc_g2, g2_z = zero
  g2_polygon_segmentg2_num_polygons, i_inc_g2 = int_unused
end do

@m increment_num_aux_sectors num_aux_sectors++
dim_aux_sectors = max(dim_aux_sectors, num_aux_sectors)
var_realloc(aux_stratum)
var_realloc(aux_stratum_poly)
var_realloc(aux_stratum_points)
var_realloc(aux_stratum_segment)
aux_stratumnum_aux_sectors = int_uninit
aux_stratum_polynum_aux_sectors = int_uninit
aux_stratum_pointsnum_aux_sectors = int_uninit
aux_stratum_segmentnum_aux_sectors = int_uninit

@m increment_num_diags num_diags++
dim_diags = max(dim_diags, num_diags)

```

```

var_realloc(a diag_name)
var_realloc(a diag_stratum)
var_realloc(a diag_solid)
var_realloc(a diag_variable)
var_realloc(a diag_tab_index)
var_realloc(a diag_var_min)
var_realloc(a diag_var_max)
var_realloc(a diag_mult)
var_realloc(a diag_spacing)
diag_name num_diags = char_uninit
diag_stratum num_diags = int_uninit
diag_solid num_diags = int_uninit
diag_variable num_diags = int_uninit
diag_tab_index num_diags = int_uninit
diag_var_min num_diags = real_uninit
diag_var_max num_diags = real_uninit
diag_mult num_diags = real_uninit
diag_spacing num_diags = int_uninit

@m open_file(aunit, aname)
open(unit = aunit, file = aname, status = 'old', form = 'formatted', iostat = open_stat)
if(open_stat ≠ 0) then
    write(stderr, *) 'Cannot open file', line(b : e), ', error number', open_stat
    return
end if

@m mesh_xzm(jxz, i, ix, iz) mesh_xz((iz-1)*nxd+ix,i,jxz) // "m" for macro.
/* The values of these macros embody the connectivity of the mesh edges. This fact is exploited in
   the corresponding code. Hence, these should not be changed without considerable thought. */
@m mesh_south 1
@m mesh_east 2
@m mesh_north 3
@m mesh_west 4
@m mesh_num_edges 4 // I.e., should equal last of the above!

```

⟨ Functions and subroutines 8 ⟩ ≡

```

program definegeometry2d
implicit none_f77
implicit none_f90
character*FILELEN filename

call readfilenames

call command_arg(1, filename)
call nc_read_materials
call def_geom_2d_main(filename)

stop
end

```

See also sections 9, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, and 35.

This code is used in section 7.1.

## 9 Main Subroutine

```

⟨ Functions and subroutines 8 ⟩ +≡
subroutine def_geom_2d_main(filename)

define_dimen(node_ind, dim_nodes)
define_dimen(element_ind, dim_elements)
define_dimen(element_ends_ind, element_start, element_end)
define_dimen(dg_poly_ind, dim_dg_poly)
define_dimen(dg_poly_meshcon_ind, meshcon_max)
define_dimen(dg_poly_meshcon_elem_ind, meshcon_elem_max)
    /* Add "2d" to distinguish from wall_ind in sector.hweb. */
define_dimen(wall2d_ind, dim_walls)
define_dimen(aux_sector_ind, dim_aux_sectors)
define_dimen(diags_ind, dim_diags)
define_dimen(mesh_corner_ind, 0, 4)
define_dimen(mesh_xz_ind, nxd * nzd)
define_dimen(mesh_nodes_ind, 4 * (nxd + nzd))
define_dimen(mesh_elements_ind, 2 * (nxd + nzd))
define_dimen(mesh_edge_elements_ind, nx_nz_max)
define_dimen(mesh_senw_ind, mesh_num_edges)
define_dimen(poly_int_ind, poly_int_max)
define_dimen(poly_real_ind, poly_real_max)
define_dimen(y_ind, 0, dim_ym)
define_dimen(y_indm, 0, dim_y - 1)
define_dimen(y_sect_ind, dim_y)

define_varp(nodes, FLOAT, g2_xz_ind, node_ind)
define_varp(node_type, INT, node_ind)
define_varp(node_element_count, INT, node_ind)

define_varp(dg_elements_list, INT, element_ends_ind, element_ind)
define_varp(element_missing, INT, element_ind)
define_varp(element_skipped, INT, element_ind)
define_varp(element_assigned, INT, element_ind)

define_varp(dg_polygons, INT, g2_points_ind, dg_poly_ind)
define_varp(dg_poly_num_elements, INT, dg_poly_ind)
define_varp(dg_poly_num_meshcon, INT, dg_poly_ind)
define_varp(dg_poly_meshcon, INT, dg_poly_meshcon_elem_ind, dg_poly_meshcon_ind, dg_poly_ind)
define_varp(dg_poly_meshcon_hv, INT, dg_poly_meshcon_elem_ind, dg_poly_meshcon_ind, dg_poly_ind)

define_varp(wall_nodes, INT, g2_points_ind0, wall2d_ind)
define_varp(wall_elements, INT, g2_points_ind, wall2d_ind)
define_varp(wall_segment_count, INT, wall2d_ind)

define_varp(y_values, FLOAT, y_ind)
define_varp(facearray, INT, y_ind)
define_varp(zonearray, INT, y_indm)
define_varp(zone_type_array, CHAR, string, y_indm)
define_varp(sect_zone1, INT, g2_poly_ind)
define_varp(sect_zone2, INT, g2_poly_ind)

define_varp(aux_stratum, INT, aux_sector_ind)
define_varp(aux_stratum_poly, INT, aux_sector_ind)

```

```

define_varp(aux_stratum_points, INT, aux_sector_ind)
define_varp(aux_stratum_segment, INT, aux_sector_ind)

define_varp(diag_name, CHAR, sc_diag_name_string, diags_ind)
define_varp(diag_stratum, INT, diags.ind)
define_varp(diag_solid, INT, diags.ind)
define_varp(diag_variable, INT, diags.ind)
define_varp(diag_tab_index, INT, diags.ind)
define_varp(diag_var_min, FLOAT, diags.ind)
define_varp(diag_var_max, FLOAT, diags.ind)
define_varp(diag_mult, FLOAT, diags.ind)
define_varp(diag_spacing, INT, diags.ind)

/* Since we can have only one adjustable dimension with pointers, combine the two into a single one
   here. We can use the desired 3-D representation in subroutines accessing these arrays. */

define_varp(mesh_xz, FLOAT, g2_xz_ind, mesh_corner_ind, mesh_xz_ind)
define_varp(mesh_nodes, FLOAT, g2_xz_ind, mesh_nodes_ind)
define_varp(mesh_elements, INT, element_ends_ind, mesh_elements_ind)
define_varp(mesh_edge_elements, INT, mesh_senw_ind, mesh_edge_elements_ind)
define_varp(mesh_edge_dg_label, INT, mesh_senw_ind, mesh_edge_elements_ind)
define_varp(mesh_edge_hv, INT, mesh_senw_ind)
define_varp(mesh_curve_num, INT, mesh_elements_ind)
define_varp(mesh_scratch, INT, mesh_elements_ind)

define_varp(poly_int_props, INT, poly_int.ind, g2_poly.ind)
define_varp(poly_real_props, FLOAT, poly_real.ind, g2_poly.ind)

implicit none_f77
gi_common
zn_common
g2_common
sc_common
ma_common
implicit none_f90

character*FILELEN filename /* Input /* Local variables, by section:
integer diskin2, length, p, b, e, open_stat, zone, fileid, /* Main section */
      prep_done, i_inc_g2, i_prop, nxpt
integer current_int_props_1:poly_int_max
real rtest
real current_real_props_1:poly_real_max
character*LINELEN line, keyword
character*FILELEN tmpfilename

integer ielement, section, i, inode, /* Process DG File */
      miss_elem, start, end, iwall, iseg, ielement2, skip_elem, poly_elem, num_h_elems,
      num_mesh_elems, mesh_elem, exp_inc
real temp_x, temp_z, mult

integer mesh_sense, ix, iz, n, mesh_pt, /* Mesh Polygons */
      poly_pt, start_pt, end_pt, inc_pt
logical new_pt

vc_decl(yhat)
vc_decl(test_vec_1)
vc_decl(test_vec_2)
vc_decl(test_vec_3)

```

```

vc_decl(center)
real x_min, x_max, z_min, z_max, xb_min, xb_max, /* Min., Max.*/
      yb_min, yb_max, zb_min, zb_max, vol

vc_decl(min_corner)
vc_decl(max_corner)

integer nxd_0, nzd_0, ix_min, ix_max, iz_min, iz_max, /* Sonnet mesh */
       nx_nz_max, mesh_tot_elements, iedge, itip, new_node, mesh_tot_nodes
integer mesh_edge_num_elements mesh_num_edges, mesh_ix_start mesh_num_edges,
       mesh_ix_end mesh_num_edges, mesh_iz_start mesh_num_edges,
       mesh_iz_end mesh_num_edges, mesh_ix_step mesh_num_edges, mesh_iz_step mesh_num_edges,
       mesh_corner mesh_num_edges, element_start:element_end
real test_node_g2_x:g2_z

integer nunit // Print Walls
character*LINELEN file_format
character*FILELEN walloutfile

integer num_new_walls, this_node, with_sonnet // Wall File
real x_wall, z_wall, xz_tmp
character*FILELEN wallinfile
character*17 exp_string

integer symmetry // Symmetry

integer solid_faces0:1, solid_zone0:0, /* Y resolution */
       end_faces0:1
character*LINELEN one_zone_type0:0
real coeff NCOEFFS, poly4_0:4,g2_x:g2_z
real y_border, cos_y, sin_y
real solid_ys0:1

vc_decl(a_y)
vc_decl(b_y)

integer process_polygon, ntriangles, refine, j, k, /* Zone, Polygon */
       temp_num_stratum_pts, temp_aux_stratum, temp_aux_segment, i_aux, nt, i_2, i_2_range, i_2_loop
integer temp_int_props1:poly_int_max, temp_segment0:max_triangles-1,0:3,
       temp_stratum_pts0:g2_num_points-1
real temp_polygon0:g2_num_points-1,g2_x:g2_z, temp_triangles0:max_triangles-1,0:3,g2_x:g2_z,
       temp_real_props1:poly_real_max, temp_holes0:0,g2_x:g2_z
character*LINELEN new_zone_type, new_zone_type_2

integer write_poly_nc // polygon.nc file
character*FILELEN polygon_nc_file

integer face1, num_zone1, num_zone2, k_zone1, /* Mixed zone sectors */
       k_zone2, solid_zone_p, solid_face, other_zone, other_face, this_poly, i_poly, this_seg, i_sect,
       solid_sector, other_sector, other_type, y_max, this_stratum, this_mat, new_solid_sector,
       new_other_sector, other_stratum, other_seg
real this_temp, this_rc

integer i_diag, i_grp, is_aux_sector // Diagnostic init
integer grp_sectors max_grp_sectors

integer y_mat0:1, y_stratum0:1, y_p_stratum0:1, end_zones0:1

```

```

integer num_nodes, num_elements, num_walls,      /* Local pointers */
        dim_nodes, dim_elements, dim_walls, nxd, nzd, num_y, y_div, dim_y, dim_ym, num_aux_sectors,
        dim_aux_sectors, num_diags, dim_diags, num_dg_poly, dim_dg_poly

declare_varp(nodes)
declare_varp(node_type)
declare_varp(node_element_count)

declare_varp(dg_elements_list)
declare_varp(element_missing)
declare_varp(element_skipped)
declare_varp(element_assigned)

declare_varp(dg_polygons)
declare_varp(dg_poly_num_elements)
declare_varp(dg_poly_num_meshcon)
declare_varp(dg_poly_meshcon)
declare_varp(dg_poly_meshcon_hv)

declare_varp(wall_nodes)
declare_varp(wall_elements)
declare_varp(wall_segment_count)

declare_varp(y_values)
declare_varp(facearray)
declare_varp(zonearray)
declare_varp(zone_type_array)
declare_varp(sect_zone1)
declare_varp(sect_zone2)

declare_varp(aux_stratum)
declare_varp(aux_stratum_poly)
declare_varp(aux_stratum_points)
declare_varp(aux_stratum_segment)

declare_varp(diag_name)
declare_varp(diag_stratum)
declare_varp(diag_solid)
declare_varp(diag_variable)
declare_varp(diag_tab_index)
declare_varp(diag_var_min)
declare_varp(diag_var_max)
declare_varp(diag_mult)
declare_varp(diag_spacing)

declare_varp(mesh_xz)
declare_varp(mesh_nodes)
declare_varp(mesh_elements)
declare_varp(mesh_edge_elements)
declare_varp(mesh_edge_dg_label)
declare_varp(mesh_edge_hv)
declare_varp(mesh_curve_num)
declare_varp(mesh_scratch)

declare_varp(poly_int_props)
declare_varp(poly_real_props)

⟨ Memory allocation interface 0 ⟩

```

```

st_decls
vc_decls
nc_decls
g2_ncdecl
gi_ext /* PREPARATION STAGE */
open(unit = diskin, file = filename, status = 'old', form = 'formatted')
diskin2 = diskin + 1

⟨ Allocations Initializations 9.1 ⟩

loop1: continue

assert(read_string(diskin, line, length))
assert(length ≤ len(line))
length = parse_string(line(: length))
p = 0
assert(next_token(line, b, e, p))
keyword = line(b : e)

if (keyword ≡ 'dg_file') then
  ⟨ Process DG File 10 ⟩

else if (keyword ≡ 'sonnet_mesh' ∨ keyword ≡ 'uedge_mesh') then
  assert(next_token(line, b, e, p))
  tmpfilename = line(b : e) // Sun F90 chokes if don't do this
  open_file(diskin2, tmpfilename) /* E.g., sonnet_mesh filename 120 24 */
  if (keyword ≡ 'sonnet_mesh') then
    assert(next_token(line, b, e, p))
    nxd_0 = read_integer(line(b : e))
    assert(next_token(line, b, e, p))
    nzd_0 = read_integer(line(b : e))
    if (next_token(line, b, e, p)) then
      ix_min = read_integer(line(b : e))
      assert(next_token(line, b, e, p))
      ix_max = read_integer(line(b : e))
      assert(next_token(line, b, e, p))
      iz_min = read_integer(line(b : e))
      assert(next_token(line, b, e, p))
      iz_max = read_integer(line(b : e))
      nxd = ix_max - ix_min + 1
      nzd = iz_max - iz_min + 1
    else
      nxd = nxd_0
      nzd = nzd_0
      ix_min = 1
      ix_max = nxd
      iz_min = 1
      iz_max = nzd
    end if
  else if (keyword ≡ 'uedge_mesh') then
    read(diskin2, *) // First line is a comment
    read(diskin2, *) nxd, nzd, nxpt
  end if
  var_alloc(mesh_xz)
  var_alloc(mesh_nodes)

```

```

var_alloc(mesh_elements)
nx_nz_max = max(nx_d, nz_d)
var_alloc(mesh_edge_elements)
var_alloc(mesh_edge_dg_label)
var_alloc(mesh_edge_hv)
var_alloc(mesh_curve_num)
var_alloc(mesh_scratch)

if (keyword ≡ 'sonnet_mesh') then
  call read_sonnet_mesh(diskin2, nx_d_0, nz_d_0, ix_min, ix_max, iz_min, iz_max, nx_d, nz_d,
                        mesh_xz)
else if (keyword ≡ 'uedge_mesh') then
  call read_uedge_mesh(diskin2, nx_d, nz_d, nxpt, mesh_xz)
  nx_d_0 = nx_d
  nz_d_0 = nz_d
  ix_min = 1
  ix_max = nx_d
  iz_min = 1
  iz_max = nz_d
end if

else if (keyword ≡ 'wallfile') then
  ⟨ Read Wall File 11 ⟩

else if (keyword ≡ 'print_min_max') then
  call find_min_max(num_nodes, nodes, node_type, x_min, x_max, z_min, z_max)
  write(stdout, *) 'Xrange', x_min, '→', x_max
  write(stdout, *) 'Zrange', z_min, '→', z_max

else if (keyword ≡ 'bounds') then
  ⟨ Set Bounds 9.2 ⟩

else if (keyword ≡ 'symmetry') then
  ⟨ Set Symmetry 9.3 ⟩

else if (keyword ≡ 'print_walls') then
  assert(next_token(line, b, e, p))
  file_format = line(b : e)
  if (next_token(line, b, e, p)) then
    walloutfile = line(b : e)
    nunit = diskout
  else
    walloutfile = char_undef
    nunit = stdout
  end if
  call print_walls(nunit, file_format, walloutfile, num_walls, wall_segment_count, wall_nodes, nodes)

else if (keyword ≡ 'uniform_y') then
  if ((symmetry ≡ geometry_symmetry_plane_hw) ∨ (symmetry ≡ geometry_symmetry_cyl_section))
    then
      assert(yb_max - yb_min > zero)
      y_border = const(1., -2) * (yb_max - yb_min)
    else if (symmetry ≡ geometry_symmetry_cyl_hw) then
      y_border = zero
      assert(yb_max - yb_min ≡ two * PI)

```

```

else if ((symmetry  $\equiv$  geometry_symmetry_oned)  $\vee$  (symmetry  $\equiv$ 
    geometry_symmetry_plane)  $\vee$  (symmetry  $\equiv$  geometry_symmetry_cylindrical))
then
    assert('Cannot specify y_values with this symmetry'  $\equiv$  ' ')
else if (symmetry  $\equiv$  geometry_symmetry_none) then
    assert('Need to specify symmetry before y_values'  $\equiv$  ' ')
end if
assert(next_token(line, b, e, p))
num_y = read_integer(line(b : e))
assert(num_y > 0)
y_div = num_y - 1
dim_ym = max(dim_ym, y_div)
var_realloc(y_values)
do i = 0, y_div
    /* This is intended to break up the existing range in y into the desired number of segments.
       For cases in which we will define solid end pieces, have set y_border to a non-zero value. */
    y_values_i = (yb_min + y_border) + ((yb_max - y_border) - (yb_min + y_border)) * areal(i) / areal(y_div)
end do

else if (keyword  $\equiv$  'y_values') then
    if ((symmetry  $\equiv$  geometry_symmetry_cyl_hw)  $\vee$  (symmetry  $\equiv$  geometry_symmetry_cyl_section))
        then
            mult = PI / const(1.8, 2)
        else if (symmetry  $\equiv$  geometry_symmetry_plane_hw) then
            mult = one
        end if
        assert(next_token(line, b, e, p))
        rtest = read_real_soft_fail(line(b : e))
        if (rtest  $\equiv$  real_UNDEF) then /* If the list of y values needs to be on separate lines, the argument
            should be a file name. That file will have one value per line. Assume these are input in
            degrees and converted here to radians. */
            tmpfilename = line(b : e)
            open_file(diskin2, tmpfilename)
            y_div = -1
        y_loop: continue
        if (read_string(diskin2, line, length)) then
            assert(length  $\leq$  len(line))
            length = parse_string(line(: length))
            p = 0
            assert(next_token(line, b, e, p))
            increment_y_div
            y_values_y_div = read_real(line(b : e)) * mult
            if (y_div > 0) then
                assert(y_values_y_div > y_values_y_div - 1)
            end if
            assert(y_values_y_div  $\geq$  yb_min)
            assert(y_values_y_div  $\leq$  yb_max)
            goto y_loop
        end if
        else /* Else, all y values have to be on a single line in the main input file. */
            y_div = 0
            y_values_0 = read_real(line(b : e)) * mult
    y_loop2: continue

```

```

if (next_token(line, b, e, p) then
  increment_y_div
  y_values_y_div = read_real(line(b : e)) * mult
  if (y_div > 0) then
    assert(y_values_y_div > y_values_y_div-1)
  end if
  goto y_loop2
  end if
end if

else if (keyword ≡ 'end_prep') then
  ⟨End Prep 9.4⟩

else if (keyword ≡ 'new_zone') then
  zone ++
  assert(next_token(line, b, e, p))
  new_zone_type = line(b : e) /* Allow for mixed zone types when the geometry is resolved in the
     third dimension. The format is to read in a second zone type, new_zone_type_2, then a list of
     y values. The first zone type is understood to be the default. The list of y values (actually,
     integers between 0 and y_div-1) indicate the divisions having the second zone type. */
  if (next_token(line, b, e, p) then
    new_zone_type_2 = line(b : e)
    assert((symmetry ≡ geometry_symmetry_plane_hw) ∨ (symmetry ≡
      geometry_symmetry_cyl_hw) ∨ (symmetry ≡ geometry_symmetry_cyl_section))
    do i = 0, y_div - 1
      zone_type_array_i = new_zone_type
    end do
    new_zone_type = "mixed"
mixed_type_loop: continue
  if (next_token(line, b, e, p) then /* Will this work? May want to try using elsewhere. */
    if (line(b : e) ≠ '-') then
      i_2 = read_integer(line(b : e))
      assert((i_2 ≥ 0) ∧ (i_2 ≤ y_div - 1))
      zone_type_array_i_2 = new_zone_type_2
    else
      assert(next_token(line, b, e, p))
      i_2_range = read_integer(line(b : e))
      assert((i_2_range ≥ 0) ∧ (i_2_range ≤ y_div - 1))
      assert(i_2_range > i_2)
      do i_2_loop = i_2 + 1, i_2_range
        zone_type_array_i_2_loop = new_zone_type_2
      end do
    end if
    goto mixed_type_loop
  end if
end if

else if (keyword ≡ 'new_polygon') then
  ⟨New Polygon 14⟩

else if (keyword ≡ 'new_diagnostic') then
  assert(next_token(line, b, e, p))
  increment_num_diags
  diag_name_num_diags = line(b : )

```

```

call specify_diagnostic(diskin, diag_stratumnum_diags, diag_solidnum_diags, diag_variablenum_diags,
                         diag_tab_indexnum_diags, diag_var_minnum_diags, diag_var_maxnum_diags, diag_multnum_diags,
                         diag_spacingnum_diags)

else if (keyword ≡ 'y_min_zone') then
    assert((symmetry ≡ geometry_symmetry_plane_hw) ∨ (symmetry ≡ geometry_symmetry_cyl_section))
    assert(next_token(line, b, e, p))
    y_mat0 = ma_lookup(line(b : e))
    assert(ma_check(y_mat0))
    assert(next_token(line, b, e, p))
    y_stratum0 = read_integer(line(b : e))
    if (next_token(line, b, e, p)) then
        y_p_stratum0 = read_integer(line(b : e))
    end if

else if (keyword ≡ 'y_max_zone') then
    assert((symmetry ≡ geometry_symmetry_plane_hw) ∨ (symmetry ≡ geometry_symmetry_cyl_section))
    assert(next_token(line, b, e, p))
    y_mat1 = ma_lookup(line(b : e))
    assert(ma_check(y_mat1))
    assert(next_token(line, b, e, p))
    y_stratum1 = read_integer(line(b : e))
    if (next_token(line, b, e, p)) then
        y_p_stratum1 = read_integer(line(b : e))
    end if

else if (keyword ≡ 'polygon_nc_file') then
    assert(next_token(line, b, e, p))
    if (line(b : e) ≡ 'none' ∨ line(b : e) ≡ 'NONE') then
        write_poly_nc = FALSE
    else
        polygon_nc_file = line(b : e)
    end if

else if (keyword ≡ 'quit' ∨ keyword ≡ 'end') then
    go to eof
end if
go to loop1

eof: continue
close(unit = diskin)

var_reallocb(g2_polygon_xz)
var_reallocb(g2_polygon_segment)
var_reallocb(g2_polygon_points)
var_reallocb(g2_polygon_zone)
var_reallocb(g2_polygon_stratum)
var_reallocb(poly_int_props)
var_reallocb(poly_real_props)
var_reallocb(aux_stratum)
var_reallocb(aux_stratum_poly)
var_reallocb(aux_stratum_points)
var_reallocb(aux_stratum_segment)
var_reallocb(diag_name)
var_reallocb(diag_stratum)
var_reallocb(diag_solid)

```

```

var_reallocb(diag_variable)
var_reallocb(diag_tab_index)
var_reallocb(diag_var_min)
var_reallocb(diag_var_max)
var_reallocb(diag_mult)
var_reallocb(diag_spacing)

var_alloc(sect_zone1)
var_alloc(sect_zone2)

if (write_poly_nc ≡ TRUE) then
    fileid = nccreate(trim(polygon_nc_file), NC_CLOBBER, nc_stat)
    g2_ncdef(fileid)
    call ncendef(fileid, nc_stat)
    g2_ncwrite(fileid)
    call ncclose(fileid, nc_stat)
end if

if (keyword ≡ 'end') then
    ⟨ Setup End Zones 15 ⟩
    call boundaries_neighbors

    call setup_sectors(g2_num_polygons, g2_polygon_points, g2_polygon_xz, g2_polygon_segment,
                        g2_polygon_zone, poly_int_props, poly_real_props, num_aux_sectors, aux_stratum,
                        aux_stratum_poly, aux_stratum_points, aux_stratum_segment, y_div, sect_zone1, sect_zone2)
    ⟨ Setup Toroidally Facing Sectors 16 ⟩
    call default_diag_setup
    ⟨ Setup Sector Diagnostics 17 ⟩
    call end_sectors
    call detector_setup
    call end_detectors
    call check_geometry
    call pixel_map_test
    call write_geometry
    call erase_geometry
end if

var_free(nodes)
var_free(node_type)
var_free(node_element_count)
var_free(dg_elements_list)
var_free(element_missing)
var_free(element_skipped)
var_free(element_assigned)
var_free(wall_nodes)
var_free(wall_elements)
var_free(wall_segment_count)
var_free(dg_polygons)
var_free(dg_poly_num_elements)
var_free(dg_poly_num_meshcon)

```

```
var_free(dg_poly_meshcon)
var_free(dg_poly_meshcon_hv)
var_free(y_values)
var_free(facearray)
var_free(zonearray)
var_free(zone_type_array)
var_free(sect_zone1)
var_free(sect_zone2)
var_free(aux_stratum)
var_free(aux_stratum_poly)
var_free(aux_stratum_points)
var_free(aux_stratum_segment)
var_free(diag_name)
var_free(diag_stratum)
var_free(diag_solid)
var_free(diag_variable)
var_free(diag_tab_index)
var_free(diag_var_min)
var_free(diag_var_max)
var_free(diag_mult)
var_free(diag_spacing)
var_free(mesh_xz)
var_free(mesh_nodes)
var_free(mesh_elements)
var_free(mesh_edge_elements)
var_free(mesh_edge_dg_label)
var_free(mesh_edge_hv)
var_free(mesh_curve_num)
var_free(mesh_scratch)
var_free(poly_int_props)
var_free(poly_real_props)

return end
```

Allocations and initializations. Apparently, this code is getting too large for FWEAVE. Splitting these lines off into a separate subsection allowed it to run successfully.

```

⟨ Allocations Initializations 9.1 ⟩ ≡
  zone = 0
  g2_num_polygons = 0
  symmetry = geometry_symmetry_none
  xb_min = zero
  xb_max = zero
  yb_min = zero
  yb_max = zero
  zb_min = zero
  zb_max = zero
  nxd = 0
  nzd = 0
  num_walls = 0
  num_nodes = 0
  num_elements = 0
  num_dg_poly = 0
  num_aux_sectors = 0
  num_diags = 0 /* To permit these to be reallocated by any of the three possible macros, helps to
                  make the initial allocation of size mem_inc. */
  dim_walls = mem_inc
  dim_nodes = mem_inc
  dim_elements = mem_inc
  dim_aux_sectors = mem_inc
  dim_diags = mem_inc
  dim_dg_poly = mem_inc

  var_alloc(nodes)
  var_alloc(node_type)
  var_alloc(node_element_count)

  var_alloc(dg_elements_list)
  var_alloc(element_missing)
  var_alloc(element_skipped)
  var_alloc(element_assigned)

  var_alloc(dg_polygons)
  var_alloc(dg_poly_num_elements)
  var_alloc(dg_poly_num_meshcon)
  var_alloc(dg_poly_meshcon)
  var_alloc(dg_poly_meshcon_hw)

  var_alloc(wall_nodes)
  var_alloc(wall_elements)
  var_alloc(wall_segment_count) /* Default in purely 2-D case. */
  y_div = 1

@if 0
  dim_y = 2 * mem_inc
@else
  dim_y = mem_inc
  dim_ym = mem_inc - 1
@endif
  var_alloc(y_values)

```

```

var_alloc(facearray)
var_alloc(zonearray)
var_alloc(zone_type_array)
y_values_0 = real_unused
y_values_1 = real_unused

y_mat_0 = int_unused
y_mat_1 = int_unused
y_stratum_0 = int_unused
y_stratum_1 = int_unused
y_p_stratum_0 = int_unused
y_p_stratum_1 = int_unused
end_zones_0 = int_unused
end_zones_1 = int_unused

var_alloc(aux_stratum)
var_alloc(aux_stratum_poly)
var_alloc(aux_stratum_points)
var_alloc(aux_stratum_segment)

var_alloc(diag_name)
var_alloc(diag_stratum)
var_alloc(diag_solid)
var_alloc(diag_variable)
var_alloc(diag_tab_index)
var_alloc(diag_var_min)
var_alloc(diag_var_max)
var_alloc(diag_mult)
var_alloc(diag_spacing)

prep_done = FALSE
current_int_props poly_stratum = int_undef
current_int_props poly_material = 0
current_int_props poly_num_holes = 0
current_real_props poly_temperature = const(3., 2)
current_real_props poly_recyc_coef = one
current_real_props poly_min_area = const(1., -3)
current_real_props poly_hole_x = real_undef
current_real_props poly_hole_z = real_undef
write_poly_nc = TRUE
polygon_nc_file = 'polygon.nc',

```

This code is used in section 9.

Set dimensions of universal cell. For cylindrical cases, the  $yb$  values are input in degrees and converted to radians at the end of the preparation section. Also, assumes that symmetry has already been set. DOCUMENT!!!

```

⟨ Set Bounds 9.2 ⟩ ≡
  assert(next_token(line, b, e, p))
  xb_min = read_real(line(b : e))
  assert(next_token(line, b, e, p))
  xb_max = read_real(line(b : e))
  assert(next_token(line, b, e, p))
  zb_min = read_real(line(b : e))
  assert(next_token(line, b, e, p))
  zb_max = read_real(line(b : e))
  if (next_token(line, b, e, p)) then
    yb_min = read_real(line(b : e))
    assert(next_token(line, b, e, p))
    yb_max = read_real(line(b : e))
  else
    if ((symmetry ≡ geometry_symmetry_plane) ∨ (symmetry ≡ geometry_symmetry_oned)) then
      /* Choose these since default source location is  $y = 0$ . */
      yb_min = -one
      yb_max = one
    else if ((symmetry ≡ geometry_symmetry_cylindrical) ∨ (symmetry ≡ geometry_symmetry_cyl_hw))
      then
        yb_min = zero
        yb_max = const(3.6, 2)
    else /* The remaining cases should have explicitly specified  $y$  bounds, but keep these since this is
           what we had before. */
      yb_min = zero
      yb_max = one
    end if
  end if
  if ((symmetry ≡ geometry_symmetry_cylindrical) ∨ (symmetry ≡
    geometry_symmetry_cyl_hw) ∨ (symmetry ≡ geometry_symmetry_cyl_section))
    then
      yb_min *= PI / const(1.8, 2)
      yb_max *= PI / const(1.8, 2)
  else
    assert((symmetry ≡ geometry_symmetry_plane) ∨ (symmetry ≡
      geometry_symmetry_plane_hw) ∨ (symmetry ≡ geometry_symmetry_oned))
  end if

```

This code is used in section 9.

Set symmetry parameter.

```
( Set Symmetry 9.3 ) ≡  
    assert(next_token(line, b, e, p))  
    if (line(b : e) ≡ 'cylindrical') then  
        symmetry = geometry_symmetry_cylindrical  
    else if (line(b : e) ≡ 'plane') then  
        symmetry = geometry_symmetry_plane  
    else if (line(b : e) ≡ 'oned') then  
        symmetry = geometry_symmetry_oned  
    else if (line(b : e) ≡ 'plane_hw') then  
        symmetry = geometry_symmetry_plane_hw  
    else if (line(b : e) ≡ 'cylindrical_hw') then  
        symmetry = geometry_symmetry_cyl_hw  
    else if (line(b : e) ≡ 'cylindrical_section') then  
        symmetry = geometry_symmetry_cyl_section  
    else  
        assert('Unexpected_symmetry_specification' ≡ '_')  
end if
```

This code is used in section 9.

End preparation stage.

$\langle$  End Prep 9.4  $\rangle \equiv$

```

dim_elements = num_elements
var_reallocb(dg_elements_list)
var_reallocb(element_missing)
var_reallocb(element_skipped)
var_reallocb(element_assigned)

dim_nodes = num_nodes
var_reallocb(nodes)
var_reallocb(node_type)
var_reallocb(node_element_count) /* Allow the "outer" points to be used in the DG polygons. To
ensure consistency of the surfaces generated from them with the universal cell surfaces, search the
node list for points close to the "outer" ones, resetting them as needed. */
do i = 1, num_nodes
  if (abs(nodesi,g2_x - xb_min) < geom_epsilon) then
    if (abs(nodesi,g2_z - zb_min) < geom_epsilon) then
      nodesi,g2_x = xb_min
      nodesi,g2_z = zb_min
    else if (abs(nodesi,g2_z - zb_max) < geom_epsilon) then
      nodesi,g2_x = xb_min
      nodesi,g2_z = zb_max
    end if
  else if (abs(nodesi,g2_x - xb_max) < geom_epsilon) then
    if (abs(nodesi,g2_z - zb_min) < geom_epsilon) then
      nodesi,g2_x = xb_max
      nodesi,g2_z = zb_min
    else if (abs(nodesi,g2_z - zb_max) < geom_epsilon) then
      nodesi,g2_x = xb_max
      nodesi,g2_z = zb_max
    end if
  end if
end do

dim_walls = num_walls
var_reallocb(wall_nodes)
var_reallocb(wall_elements)
var_reallocb(wall_segment_count)

dim_dg_poly = num_dg_poly
var_reallocb(dg_polygons)
var_reallocb(dg_poly_num_elements)
var_reallocb(dg_poly_num_meshcon)
var_reallocb(dg_poly_meshcon)
var_reallocb(dg_poly_meshcon_hv)

dim_y = y_div
dim_ym = y_div
var_reallocb(y_values)
var_reallocb(facearray)
var_reallocb(zonearray)
var_reallocb(zone_type_array)

call init_geometry /* yb_min and yb_max are now set with the bounds keyword. */

```

```

if (symmetry  $\equiv$  geometry_symmetry_cyl_section) then
  assert(yb_max - yb_min < two * PI)
end if
vc_set(min_corner, xb_min, yb_min, zb_min)
vc_set(max_corner, xb_max, yb_max, zb_max)
  /* Have generalized this for the various 3-D cases, following the planar case in boxgen. */
call universal_cell_3d(symmetry, min_corner, max_corner, vol)

solid_ys0 = y_values0
solid_ys1 = y_valuesy_div
solid_faces0 = int_unused
solid_faces1 = int_unused
if ((symmetry  $\equiv$  geometry_symmetry_cylindrical)  $\vee$  (symmetry  $\equiv$  geometry_symmetry_plane)) then
  facearray0 = int_unused    // Default case
  facearray1 = int_unused
else
  do i = 0, y_div
    if (symmetry  $\equiv$  geometry_symmetry_plane_hw) then
      vc_set(a_y, zero, one, zero)
      vc_set(b_y, xb_max, y_valuesi, zb_max)
    else
      assert((symmetry  $\equiv$  geometry_symmetry_cyl_hw)  $\vee$  (symmetry  $\equiv$  geometry_symmetry_cyl_section))
        /* Ensure beginning and end faces match. */
      if ((symmetry  $\equiv$  geometry_symmetry_cyl_hw)  $\wedge$  (i  $\equiv$  y_div)) then
        cos_y = cos(y_values0)
        sin_y = sin(y_values0)
      else
        cos_y = cos(y_valuesi)    // General case
        sin_y = sin(y_valuesi)
      end if
      vc_set(a_y, -sin_y, cos_y, zero)
      vc_set(b_y, xb_max * cos_y, xb_max * sin_y, zb_max)
    end if
    call plane(b_y, a_y, coeff)
    facearrayi = define_surface(coeff, T)
  end do    /* Don't think this will be enough. */
if ((symmetry  $\equiv$  geometry_symmetry_plane_hw)  $\vee$  (symmetry  $\equiv$  geometry_symmetry_cyl_section))
  then
    solid_faces0 = facearray0
    solid_faces1 = facearrayy_div
end if
end if

if (nxd * nzd > 0) then
  ⟨ Set Mesh Polygons 12 ⟩
  ⟨ Set Mesh Elements 13 ⟩
else    /* Allocate something since it appears in argument lists later. */
  nxd = 1
  nzd = 1
  var_alloc(mesh_xz)
  var_alloc(mesh_nodes)
  var_alloc(mesh_elements)
  nx_nz_max = 1

```

```
var_alloc(mesh_edge_elements)
var_alloc(mesh_edge_dg_label)
var_alloc(mesh_edge_hv)
var_alloc(mesh_curve_num)
var_alloc(mesh_scratch)
end if
prep_done = TRUE
```

This code is used in section 9.

## 10 Read and process DG file

```

⟨ Process DG File 10 ⟩ ≡
  assert(next_token(line, b, e, p))
  tmpfilename = line(b : e)
  open_file(diskin2, tmpfilename) /* Read DG's .dgo file */
  ielement = 0
  section = sec_undef /* Assume that if the file has a ".dgo" that it came from DG and needs to be
                       converted from mm to meters. Assume that it is otherwise in meters. Developed read_real_scaled
                       expressly for this task since direct multiplication introduces round-off error that can cause these
                       coordinates to differ from the same ones read in from other files. */
  if(index(line(b : e), '.dgo') > 0) then
    exp_inc = -3
  else
    exp_inc = 0
  end if

dg_loop: continue
if(read_string(diskin2, line, length)) then
  assert(length ≤ len(line))
  /* We do not want to have to manually change the format of the files DG writes. The node
   coordinates are currently comma-delimited (with possible additional spaces). Replace the
   commas in the current line with spaces so we can use our usual string utilities. */
  do i = 1, length
    if(line(i : i) ≡ ',')
      line(i : i) = ' '
  end do
  length = parse_string(line(: length))
  p = 0
  assert(next_token(line, b, e, p))
  if(line(b : e) ≡ 'p1') then
    section = sec_p1
  else if(line(b : e) ≡ 'p2') then
    section = sec_p2
  else if(line(b : e) ≡ 'misselem') then
    section = sec_miss
  else if(line(b : e) ≡ 'skipelem') then
    section = sec_skip
  else if(line(b : e) ≡ 'polygon') then
    assert(next_token(line, b, e, p))
    increment_num_dg_poly
    assert(num_dg_poly ≡ read_integer(line(b : e)))
    dg_poly_num_elements_num_dg_poly = 0
    dg_poly_num_meshcon_num_dg_poly = 0
    do i = 1, meshcon_max
      do j = 1, meshcon_elem_max
        dg_poly_meshcon_num_dg_poly,i,j = int_unused
        dg_poly_meshcon_hv_num_dg_poly,i,j = int_unused
      end do
    end do
    section = sec_dg_poly
  else if(line(b : e) ≡ 'wall') then

```

```

assert(section ≡ sec_dg_poly)
section = sec_dg_wall
    /* Note that we are assuming we know the order of the polygon variables. Since we have
       explicitly defined them in DG, this should be OK. The "polymat" entry is a single line, so
       does not need a separate section and would have to process here. Not using it yet, though. */
else if(line(b : e) ≡ 'polymat') then
    assert(section ≡ sec_dg_wall)
else if(line(b : e) ≡ 'meshcon1') then
    assert(section ≡ sec_dg_wall)
    section = sec_dg_meshcon1
else if(line(b : e) ≡ 'meshcon2') then
    assert(section ≡ sec_dg_meshcon1)
    section = sec_dg_meshcon2
    /* ADD SECTION TO READ jedgi1, jedgo2, jedgo1, jedgi2 SECTIONS AND PROCESS */
else if(line(b : e) ≡ 'finish') then
    section = sec_done
else
    if(section ≡ sec_p1 ∨ section ≡ sec_p2) then
        temp_x = read_real_scaled(line(b : e), exp_inc)
        if(temp_x ≠ real_undef) then    // Failure
            assert(next_token(line, b, e, p))
            temp_z = read_real_scaled(line(b : e), exp_inc)
        end if
        if(temp_x ≡ real_undef ∨ temp_z ≡ real_undef) then
            section = sec_undef
            go to dg_loop    // Can trash this line since it did not match anything we've planned for
        end if
    @#if 0
        // Replaced this with the use of read_real_scaled .
        temp_x *= mult
        temp_z *= mult
    @#endif
        inode = 0
        if(num_nodes > 0) then
            do i = 1, num_nodes
                if(nodesi,g2_x ≡ temp_x) then
                    if(nodesi,g2_z ≡ temp_z) then
                        assert(inode ≡ 0)
                        inode = i    // is a duplicate
                    end if
                end if
            end do
        end if
        if(inode ≡ 0) then
            increment_num_nodes    // is a new node
            inode = num_nodes
            nodesnum_nodes,g2_x = temp_x
            nodesnum_nodes,g2_z = temp_z
        end if
        if(section ≡ sec_p1) then
            increment_num_elements
            dg_elements_listnum_elements,element_start = inode

```

```

element_missing_num_elements = FALSE      // will be set later
element_skipped_num_elements = FALSE      // ditto
element_assigned_num_elements = FALSE
else if (section ≡ sec_p2) then
    ielement++
    dg_elements_listielement,element_end = inode
end if /* Note that "missing" elements are those that should be completely ignored, e.g.,
elements that were deleted, but still appear in the DG file. "Skipped" elements, on
the other hand, are intended to be used in setting up DG-defined polygons, but not in
characterizing nodes or setting up walls. */
else if (section ≡ sec_miss) then
    assert(ielement ≡ num_elements) // Compare p1 and p2 reads
    miss_elem = read_int_soft_fail(line(b : e))
    if (miss_elem > 0 ∧ miss_elem ≤ num_elements) then
        element_missingmiss_elem = TRUE
    else
        assert(miss_elem ≡ int_undef) // Assume we've stumbled into another section
        section = sec_undef
        go to dg_loop
    end if
else if (section ≡ sec_skip) then
    assert(ielement ≡ num_elements) // Compare p1 and p2 reads
    skip_elem = read_int_soft_fail(line(b : e))
    if (skip_elem > 0 ∧ skip_elem ≤ num_elements) then
        element_skippedskip_elem = TRUE
    else
        assert(skip_elem ≡ int_undef) // Assume we've stumbled into another section
        section = sec_undef
        go to dg_loop
    end if
else if (section ≡ sec_dg_wall) then
    poly_elem = read_int_soft_fail(line(b : e))
    assert((poly_elem > 0) ∧ (poly_elem ≤ num_elements))
    assert(num_dg_poly > 0)
    dg_poly_num_elementsnum_dg_poly++
    dg_polygonsnum_dg_poly,dg_poly_num_elements num_dg_poly = poly_elem
else if ((section ≡ sec_dg_meshcon1) ∨ (section ≡ sec_dg_meshcon2)) then
    num_h_elems = read_int_soft_fail(line(b : e))
    assert((num_h_elems ≥ 0) ∧ (num_h_elems ≤ 2))
    num_mesh_elems = 0 /* Processing of these sections differs from the others in that we have a
count (first line) and then a list of elements, which is at least "count" long. Because of this,
we need to process the list here. Have set this up so that either one of the two "meshcon"
variables can be used to specify a single point. */
meshcon_loop: continue
    assert(read_string(diskin2, line, length))
    assert(length ≤ len(line))
    length = parse_string(line(:length))
    p = 0
    assert(next_token(line, b, e, p))
    mesh_elem = read_int_soft_fail(line(b : e))
    if (mesh_elem ≠ int_undef) then
        if (num_mesh_elems ≡ 0) then

```

```

    dg_poly_num_meshcon_num_dg_poly ++
    assert(dg_poly_num_meshcon_num_dg_poly ≤ meshcon_max)
end if
num_mesh_elems ++
assert(num_mesh_elems ≤ meshcon_elem_max)
dg_poly_meshcon_num_dg_poly,dg_poly_num_meshcon_num_dg_poly,num_mesh_elems = mesh_elem
if (num_mesh_elems ≤ num_h_elems) then
    dg_poly_meshcon_hv_num_dg_poly,dg_poly_num_meshcon_num_dg_poly,num_mesh_elems = mesh_h
else
    dg_poly_meshcon_hv_num_dg_poly,dg_poly_num_meshcon_num_dg_poly,num_mesh_elems = mesh_v
end if
go to meshcon_loop
else /* The list of elements is finished. Use the backspace command here to allow the main
       loop to read this line again and handle accordingly. */
backspace(diskin2)
assert(num_mesh_elems ≥ num_h_elems)
assert((num_mesh_elems ≡ 0) ∨ (num_mesh_elems ≡ meshcon_elem_max))
/* This is the last piece of the polygon section. Reset section since we don't know what
   might be next. */
if (section ≡ sec_dg_meshcon2)
    section = sec_undef
go to dg_loop
end if
end if
end if
if (section ≠ sec_done)
    go to dg_loop
end if
close(unit = diskin2) /* Now have elements and nodes. Characterize all of the nodes. */
do inode = 1, num_nodes
    node_element_count_inode = 0
    start = FALSE end = FALSE
    do ielement = 1, num_elements
        if ((element_missing_ielement ≡ FALSE) ∧ (element_skipped_ielement ≡ FALSE)) then
            if (dg_elements_list_ielement,element_start ≡ inode) then
                node_element_count_inode ++
                start = TRUE
            else if (dg_elements_list_ielement,element_end ≡ inode) then
                node_element_count_inode ++ end = TRUE
            end if
        end if
    end do
    if (node_element_count_inode ≡ 0) then
        node_type_inode = node_no_elements
    else if (node_element_count_inode ≡ 1) then assert ( start ≡ TRUE ∨ end ≡ TRUE )
        node_type_inode = node_one_element
    else if (node_element_count_inode ≡ 2) then if ( start ≡ TRUE ∧ end ≡ TRUE ) then
        node_type_inode = node_regular
    else
        node_type_inode = node_mixed_normals
    end if

```

```

else if (node_element_countinode > 2) then
    node_typeinode = node_many_elements
end if
end do /* Set up the walls. First, start walls at all irregular nodes that start an element. */
do inode = 1, num_nodes
if (node_typeinode ≠ node_regular ∧ node_typeinode ≠ node_no_elements) then
    do ielement = 1, num_elements
        if (dg_elements_listielement,element_start ≡ inode) then
            increment_num_walls
            wall_elementsnum_walls,1 = ielement
            wall_nodesnum_walls,0 = inode
            element_assignedielement = TRUE
        end if
    end do
end if
end do /* Next, follow each of those walls until they hit another irregular node. */
do iwall = 1, num_walls
    iseg = 1
wall_loop: continue
    inode = dg_elements_listwall_elementsiwall,iseg,element_end
    assert (inode > 0 ∧ inode ≤ num_nodes)
    wall_nodesiwall,iseg = inode
    if (node_typeinode ≡ node_regular) then
        iseg ++
        assert (iseg ≤ g2_num_points)
        wall_elementsiwall,iseg = 0
        do ielement = 1, num_elements
            if ((dg_elements_listielement,element_start ≡ inode) ∧ (element_skippedielement ≡ FALSE)) then
                assert (wall_elementsiwall,iseg ≡ 0)
                wall_elementsiwall,iseg = ielement
                element_assignedielement = TRUE
                go to wall_loop
            end if
        end do
    else // End the wall at the next irregular node.
        wall_segment_countiwall = iseg
    end if
end do /* Finally, follow all other elements connected only by regular nodes (these should be closed surfaces). */
do ielement = 1, num_elements
if ((element_missingielement ≡ FALSE) ∧ (element_skippedielement ≡ FALSE) ∧ (element_assignedielement ≡ FALSE)) then
    increment_num_walls // An unassigned element...
    wall_elementsnum_walls,1 = ielement
    wall_nodesnum_walls,0 = dg_elements_listielement,element_start
    element_assignedielement = TRUE
    iseg = 1
wall_loop2: continue
    inode = dg_elements_listwall_elementsnum_walls,iseg,element_end
    assert (inode > 0 ∧ inode ≤ num_nodes)
    wall_nodesnum_walls,iseg = inode

```

```

assert(node_typeinode ≡ node_regular)
/* Search for adjacent element. Can start at ielement+1 because all previous elements have been
assigned. If it has already been assigned, we're done with this wall. */
do ielement2 = ielement + 1, num_elements
  if ((dg_elements_listielement2, element_start ≡ inode) ∧ (element_assignedielement2 ≡ FALSE)) then
    iseg++
    assert(iseg ≤ g2_num_points)
    wall_elementsnum_walls, iseg = ielement2
    element_assignedielement2 = TRUE
    go to wall_loop2
  end if
end do
wall_segment_countnum_walls = iseg // End of this wall

end if
end do

@if 0
  // Move to after end of prep. stage. OK?
  var_reallocb(wall_nodes)
  var_reallocb(wall_elements)
  var_reallocb(wall_segment_count)
#endif

```

This code is used in section 9.

## 11 Read wall file

This is intended to duplicate the wallfile functionality provided in *readgeometry*. At the same time, this format should allow a file generated with the “linear” format of this code’s *print\_walls* keyword to be read back in. Hence, the numbers on the second line of the file represent the **number of points, not the number of segments**, as was the case in *readgeometry*. The points listed are checked against the existing list of nodes. If new, they are added. Either way, the new walls are stored as a list of nodes, just as are those generated from a DG file. Note that we presently make no attempt to update the elements. Only the node numbers and coordinates are used later in the code.

$\langle \text{Read Wall File 11} \rangle \equiv$

```

assert(next_token(line, b, e, p))
wallinfile = line(b : e)
if (next_token(line, b, e, p)) then
    assert(line(b : e) ≡ 'with_sonnet')
    with_sonnet = TRUE
else
    with_sonnet = FALSE
end if
open(unit = diskin2, file = wallinfile, status = 'old', form = 'formatted')
assert(read_string(diskin2, line, length))
assert(length ≤ len(line))
length = parse_string(line(: length))
p = 0
assert(next_token(line, b, e, p))
num_new_walls = read_int_soft_fail(line(b : e))
var_realloc(wall_nodes, dim_walls, num_walls + num_new_walls)
var_realloc(wall_elements, dim_walls, num_walls + num_new_walls) // not used
var_realloc(wall_segment_count, dim_walls, num_walls + num_new_walls)

assert(read_string(diskin2, line, length))
assert(length ≤ len(line))
length = parse_string(line(: length))
p = 0
do iwall = 1, num_new_walls
    if (¬next_token(line, b, e, p)) then
        assert(read_string(diskin2, line, length))
        assert(length ≤ len(line))
        length = parse_string(line(: length))
        p = 0
        assert(next_token(line, b, e, p))
    end if
    wall_segment_countnum_walls+iwall = read_int_soft_fail(line(b : e))
end do

do iwall = num_walls + 1, num_walls + num_new_walls
    wall_segment_countiwall-- // iseg starts at 0
    do iseg = 0, wall_segment_countiwall
        assert(read_string(diskin2, line, length))
        assert(length ≤ len(line))
        length = parse_string(line(: length))
        p = 0
    end do

```

```

assert(next_token(line, b, e, p))
/* Using read_real_soft_fail here since debugging read statements is difficult. Instead, use this
   assertion to check for the error indicator returned by read_real_soft_fail. */
xz_tmp = read_real_soft_fail(line(b : e))
if (with_sonnet ≡ TRUE) then /* This process is intended to mimic the action performed by
   tri_to_sonnet in reading (arbitrarily formatted) Triangle format files and writing them out
   in Sonnet format. The e17.10 format is precisely the one used there, as well as in the
   read_sonnet_mesh routine. */
  write(exp_string, '(e17.10)') xz_tmp
  read(exp_string, '(e17.10)') x_wall
else
  x_wall = xz_tmp
end if
assert(x_wall ≠ real_undef)
assert(next_token(line, b, e, p))
xz_tmp = read_real_soft_fail(line(b : e))
if (with_sonnet ≡ TRUE) then
  write(exp_string, '(e17.10)') xz_tmp
  read(exp_string, '(e17.10)') z_wall
else
  z_wall = xz_tmp
end if
assert(z_wall ≠ real_undef)
if (num_nodes > 0) then
  do inode = 1, num_nodes
    if ((abs(x_wall - nodes(inode,g2_x)) < geom_epsilon) ∧ (abs(z_wall - nodes(inode,g2_z)) <
       geom_epsilon)) then
      this_node = inode
      go to wall_break
    end if
  end do
end if
increment_num_nodes
nodes(num_nodes,g2_x) = x_wall
nodes(num_nodes,g2_z) = z_wall
this_node = num_nodes

wall_break: continue
  wall_nodes(iwall,iseg) = this_node
end do
end do
num_walls += num_new_walls
dim_walls = max(dim_walls, num_walls)
close(unit = diskin2)

```

This code is used in section 9.

## 12 Convert mesh data into polygons

```

⟨ Set Mesh Polygons 12 ⟩ ≡
  /* A separate question is the direction around the physical rectangle. The polygon sent to
   decompose_polygon must be traversed in a clockwise direction in physical space. */
  vc_set(yhat, zero, one, zero)
  vc_set(test_vec_1, mesh_xzm(g2_x, 2, 1, 1) - mesh_xzm(g2_x, 1, 1, 1), zero, mesh_xzm(g2_z, 2, 1,
    1) - mesh_xzm(g2_z, 1, 1, 1))
  vc_set(test_vec_2, mesh_xzm(g2_x, 3, 1, 1) - mesh_xzm(g2_x, 2, 1, 1), zero, mesh_xzm(g2_z, 3, 1,
    1) - mesh_xzm(g2_z, 2, 1, 1))
  vc_cross(test_vec_1, test_vec_2, test_vec_3)
  if (vc_product(test_vec_3, yhat) > zero) then
    mesh_sense = 1
  else if (vc_product(test_vec_3, yhat) < zero) then
    mesh_sense = 2
  else
    assert('First_cell_of_mesh_degenerate' ≡ ' ')
  end if
  do ix = 1, nxd
    do iz = 1, nzd
      zone++
      n = 0
      increment_g2_num_polygons
      g2_polygon_xz_g2_num_polygons,n,g2_x = mesh_xzm(g2_x, 1, ix, iz)
      g2_polygon_xz_g2_num_polygons,n,g2_z = mesh_xzm(g2_z, 1, ix, iz)
      /* Regardless of the sense of the mesh, the points are numbered consecutively, so just need to
       alter the loop indices in transferring the points to g2_polygon_xz. */
      if (mesh_sense ≡ 1) then
        start_pt = 2
        end_pt = 4
        inc_pt = 1
      else if (mesh_sense ≡ 2) then
        start_pt = 4
        end_pt = 2
        inc_pt = -1
      else
        assert('mesh_sense_improperly_set' ≡ ' ')
      end if /* Since we also use the Sonnet mesh format for triangular meshes, identifying and
               removing duplicate points turns out to be useful. Although there is a specific convention used
               in assigning the duplicate point in the Sonnet file, explicitly checking all points, as we do
               here, is not too cumbersome. */
      do mesh_pt = start_pt, end_pt, inc_pt
        new_pt = T
        do poly_pt = 0, n
          if ((mesh_xzm(g2_x, mesh_pt, ix,
            iz) ≡ g2_polygon_xz_g2_num_polygons,poly_pt,g2_x) ∧ (mesh_xzm(g2_z, mesh_pt, ix,
            iz) ≡ g2_polygon_xz_g2_num_polygons,poly_pt,g2_z)) then
            new_pt = F
          end if
        end do
        if (new_pt) then

```

```

n++
g2_polygon_xzg2_num_polygons,n,g2_x = mesh_xzm(g2_x, mesh_pt, ix, iz)
g2_polygon_xzg2_num_polygons,n,g2_z = mesh_xzm(g2_z, mesh_pt, ix, iz)
end if
end do
n++
g2_polygon_xzg2_num_polygons,n,g2_x = g2_polygon_xzg2_num_polygons,0,g2_x
g2_polygon_xzg2_num_polygons,n,g2_z = g2_polygon_xzg2_num_polygons,0,g2_z
do i = 0, n - 1
  g2_polygon_segmentg2_num_polygons,i = i
end do
g2_polygon_segmentg2_num_polygons,n = 0 /* Now also have g2_polygon_stratum to set, but not
   clear if there is any intelligent use for it since we already have the zone number. */
g2_polygon_pointsg2_num_polygons = n
g2_polygon_zoneg2_num_polygons = zone
/* These are really not used for solid zones and will probably contain the default values. */
do i_prop = 1, poly_int_max
  poly_int_propsg2_num_polygons,i_prop = current_int_propsi_prop
end do
do i_prop = 1, poly_real_max
  poly_real_propsg2_num_polygons,i_prop = current_real_propsi_prop
end do
@if 0
zonearray0 = zone
call decompose_polygon(n, g2_polygon_xzg2_num_polygons,0,g2_x, zonearray, 1, facearray)
@else
do i = 0, y_div - 1
  if (i > 0)
    zone = zone + 1
    zonearrayi = zone
    zone_type_arrayi = "plasma"
  end do
  call decompose_polygon(n, g2_polygon_xzg2_num_polygons,0,g2_x, zonearray, y_div, facearray)
@endif
/* Check to see if the mesh center is zero. If so, set using existing routines (note: triangles can
now be specified in a Sonnet mesh format, but only if corner 4 matches corner 1 or 3). */
if ((mesh_xzm(g2_x, 0, ix, iz) ≡ zero) ∧ (mesh_xzm(g2_z, 0, ix, iz) ≡ zero)) then
  if (((mesh_xzm(g2_x, 4, ix, iz) ≡ mesh_xzm(g2_x, 1, ix, iz)) ∧ (mesh_xzm(g2_z, 4, ix,
    iz) ≡ mesh_xzm(g2_z, 1, ix, iz))) ∨ ((mesh_xzm(g2_x, 4, ix, iz) ≡ mesh_xzm(g2_x, 3,
    ix, iz)) ∧ (mesh_xzm(g2_z, 4, ix, iz) ≡ mesh_xzm(g2_z, 3, ix, iz)))) then
    call triangle_centroid(mesh_xzm(g2_x, 1, ix, iz), center)
  else
    call quad_center(mesh_xzm(g2_x, 1, ix, iz), center)
  end if
else
  vc_set(center, mesh_xzm(g2_x, 0, ix, iz), zero, mesh_xzm(g2_z, 0, ix, iz))
end if
call update_zone_info(zonearray, ix, iz, zone_type_array, n, g2_polygon_xzg2_num_polygons,0,g2_x,
  center, y_div, y_values)
enddo
enddo

```

This code is used in section 9.4.

## 13 Set up mesh elements

Convert the edges of the Sonnet or UEDGE mesh into nodes and elements analogous to those read in from the DG file. This step facilitate their use in polygons.

```
< Set Mesh Elements 13 > ==
/* These are the basic parameters describing the mesh. The directional names for the edges comes
   from an old B2 convention. The basic idea is that looping over these edges corresponds to a
   counterclockwise trip around the computational mesh. The corresponding values for the corner
   indices can be verified using the diagram provided in the introductory documentation of this code.
*/
mesh_ix_start mesh_south = 1
mesh_ix_end mesh_south = nxd
mesh_iz_start mesh_south = 1
mesh_iz_end mesh_south = 1
mesh_edge_num_elements mesh_south = nxd
mesh_ix_step mesh_south = 1
mesh_iz_step mesh_south = 0
mesh_corner mesh_south,element_start = 1
mesh_corner mesh_south,element_end = 4
mesh_edge_hv mesh_south = mesh_h

mesh_ix_start mesh_east = nxd
mesh_ix_end mesh_east = nxd
mesh_iz_start mesh_east = 1
mesh_iz_end mesh_east = nzd
mesh_edge_num_elements mesh_east = nzd
mesh_ix_step mesh_east = 0
mesh_iz_step mesh_east = 1
mesh_corner mesh_east,element_start = 4
mesh_corner mesh_east,element_end = 3
mesh_edge_hv mesh_east = mesh_v

mesh_ix_start mesh_north = nxd
mesh_ix_end mesh_north = 1
mesh_iz_start mesh_north = nzd
mesh_iz_end mesh_north = nzd
mesh_edge_num_elements mesh_north = nxd
mesh_ix_step mesh_north = -1
mesh_iz_step mesh_north = 0
mesh_corner mesh_north,element_start = 3
mesh_corner mesh_north,element_end = 2
mesh_edge_hv mesh_north = mesh_h

mesh_ix_start mesh_west = 1
mesh_ix_end mesh_west = 1
mesh_iz_start mesh_west = nzd
mesh_iz_end mesh_west = 1
mesh_edge_num_elements mesh_west = nzd
mesh_ix_step mesh_west = 0
mesh_iz_step mesh_west = -1
mesh_corner mesh_west,element_start = 2
mesh_corner mesh_west,element_end = 1
```

```

mesh_edge_hv mesh_west = mesh_v
    /* Set up the nodes and elements contained in the four (mesh_num_edges) mesh edges. */
mesh_tot_elements = 0
mesh_tot_nodes = 0
do iedge = 1, mesh_num_edges
    ix = mesh_ix_start_iedge
    iz = mesh_iz_start_iedge
    do iseg = 1, mesh_edge_num_elements_iedge      /* Each segment of a mesh edge yields a new
        (presumably!) element, with a start and an end. */
        mesh_tot_elements++
        mesh_edge_elements_iseg,iedge = mesh_tot_elements
        mesh_edge_dg_label_iseg,iedge = (iz + (iz_min - 1) - 1) * nxd_0 + (ix + (ix_min - 1) - 1)
        if (iedge ≡ mesh_east) then
            if (ix_max < nxd_0) then
                mesh_edge_dg_label_iseg,iedge ++
            else
                mesh_edge_dg_label_iseg,iedge = -(mesh_edge_dg_label_iseg,iedge + 1)
            end if
        else if (iedge ≡ mesh_north) then
            if (iz_max < nzd_0) then
                mesh_edge_dg_label_iseg,iedge += nxd_0
            else
                mesh_edge_dg_label_iseg,iedge = -(mesh_edge_dg_label_iseg,iedge + 1)
            end if
        end if
        do itip = element_start, element_end      /* The points at each end of the segment is compared
            with the list of existing nodes and added to that list if new. */
            test_node_g2_x = mesh_xzm(g2_x, mesh_corner_iedge, itip, ix, iz)
            test_node_g2_z = mesh_xzm(g2_z, mesh_corner_iedge, itip, ix, iz)
            new_node = TRUE
            if (mesh_tot_nodes > 0) then
                do inode = 1, mesh_tot_nodes
                    if ((mesh_nodes_inode,g2_x ≡ test_node_g2_x) ∧ (mesh_nodes_inode,g2_z ≡ test_node_g2_z)) then
                        assert(new_node ≡ TRUE) // Right?
                        new_node = FALSE
                        this_node = inode
                    end if
                end do
            end if
            if (new_node ≡ TRUE) then
                mesh_tot_nodes++
                mesh_nodes_mesh_tot_nodes,g2_x = test_node_g2_x
                mesh_nodes_mesh_tot_nodes,g2_z = test_node_g2_z
                this_node = mesh_tot_nodes
            end if
            mesh_elements_mesh_tot_elements,tip = this_node
        end do // itip
        ix += mesh_ix_step_iedge
        iz += mesh_iz_step_iedge
    end do // iseg
    assert(ix ≡ mesh_ix_end_iedge + mesh_ix_step_iedge)
    assert(iz ≡ mesh_iz_end_iedge + mesh_iz_step_iedge)

```

```
end do // iedge /* Note that mesh "cuts" are manifested as adjacent corners of adjacent elements
not being identical. Since this situation is the exception rather than the rule, should really have
mesh_tot_nodes < 4(nxd + nzd).
assert(mesh_tot_nodes < 4 * (nxd + nzd))
assert(mesh_tot_elements == 2 * (nxd + nzd))
```

This code is used in section 9.4.

## 14 Specify and break up a new polygon

```

⟨ New Polygon 14 ⟩ ≡
  assert(prep_done ≡ TRUE)
  call specify_polygon(diskin, num_walls, wall_segment_count, wall_nodes, nodes, dg_elements_list,
    num_dg_poly, dg_poly_num_elements, dg_polygons, dg_poly_num_meshcon, dg_poly_meshcon,
    dg_poly_meshcon_hv, xb_min, xb_max, zb_min, zb_max, nxd, nzd, mesh_xz, mesh_nodes,
    mesh_elements, mesh_edge_num_elements, mesh_edge_elements, mesh_edge_dg_label,
    mesh_edge_hv, mesh_curve_num, mesh_scratch, temp_polygon0,g2_x, n, temp_int_props,
    temp_real_props, temp_aux_stratum, temp_stratum_pts, temp_num_stratum_pts, process_polygon)
  if (process_polygon ≠ clear_polygon) then
    ix = 0
    iz = 0
    do i_prop = 1, poly_int_max
      if (temp_int_propsi_prop ≠ int_uninit)
        current_int_propsi_prop = temp_int_propsi_prop
    end do
    do i_prop = 1, poly_real_max
      if (temp_real_propsi_prop ≠ real_uninit)
        current_real_propsi_prop = temp_real_propsi_prop
    end do
  if (process_polygon ≡ breakup_polygon) then
    increment_g2_num_polygons
    do i = 0, n
      g2_polygon_xzg2_num_polygons,i,g2_x = temp_polygoni,g2_x
      g2_polygon_xzg2_num_polygons,i,g2_z = temp_polygoni,g2_z
      if (i < n) then
        g2_polygon_segmentg2_num_polygons,i = i
      else
        g2_polygon_segmentg2_num_polygons,i = 0
      end if
    end do
    g2_polygon_pointsg2_num_polygons = n
    g2_polygon_zoneg2_num_polygons = zone
    g2_polygon_stratumg2_num_polygons = current_int_propspoly_stratum
    do i_prop = 1, poly_int_max
      poly_int_propsg2_num_polygons,i_prop = current_int_propsi_prop
    end do
    do i_prop = 1, poly_real_max
      poly_real_propsg2_num_polygons,i_prop = current_real_propsi_prop
    end do
    /* Had originally thought of dividing plasma and vacuum zones up in y, but assume
       user would be using triangulate_to_zones in that case. */
    solid_zone0 = zone
    call decompose_polygon(n, g2_polygon_xzg2_num_polygons,0,g2_x, solid_zone, 1, solid_faces)
    /* Try to handle carefully those cases where we can. For n > 4, just use the first point. If this
       zone really needs a center, can break up into triangles first. */
    if (n ≡ 3) then
      call triangle_centroid(temp_polygon0,g2_x, center)
    else if (n ≡ 4) then
      call quad_center(temp_polygon0,g2_x, center)
  
```

```

else
  vc_set(center, temp_polygon0,g2_x, zero, temp_polygon0,g2_z)
end if
one_zone_type0 = new_zone_type

call update_zone_info(solid_zone, ix, iz, one_zone_type, n, g2_polygon_xzg2_num_polygons,0,g2_x,
                      center, 1, solid_ys)
/* Transfer data on auxiliary strata to main arrays. In this case, the original polygon point
   number can be used directly to identify the chosen segment in setup_sectors. */

if (temp_num_stratum_pts > 0) then
  do i = 1, temp_num_stratum_pts
    increment_num_aux_sectors
    aux_stratum_num_aux_sectors = temp_aux_stratum
    aux_stratum_poly_num_aux_sectors = g2_num_polygons
    aux_stratum_points_num_aux_sectors = temp_stratum_ptsi
    /* Assign segment number for a stratum just by counting the number that have been
       specified. Easiest to just start at the end and find the last one. */
    temp_aux_segment = 1
  if (num_aux_sectors > 1) then
    do i_aux = num_aux_sectors - 1, 1, -1
      if (aux_stratumi_aux ≡ temp_aux_stratum) then
        temp_aux_segment = aux_stratum_segmenti_aux + 1
        go to sector_break
      end if
    end do
  sector_break: continue
  end if
  aux_stratum_segment_num_aux_sectors = temp_aux_segment
  end do
end if

else if (process_polygon ≡ triangulate_polygon ∨ process_polygon ≡ triangulate_to_zones) then
  if (process_polygon ≡ triangulate_to_zones) then
    refine = TRUE // we care
  else
    refine = FALSE // we're just breaking 'em up.
  end if
  if (current_int_props.poly_num_holes > 0) then /* Not sure of the need for multiple holes.
    Easiest to implement just a single one. Note that the segment labels for the initial polygon
    are assigned in poly2triangles and transferred to the resulting triangles in temp_segment. */
    assert(current_int_props.poly_num_holes ≡ 1)
    temp_holes0,g2_x = current_real_props.poly_hole_x
    temp_holes0,g2_z = current_real_props.poly_hole_z
  end if
  call poly2triangles(n, temp_polygon0,g2_x, current_real_props.poly_min_area,
                      current_int_props.poly_num_holes, temp_holes0,g2_x, refine, ntriangles,
                      temp_triangles0,0,g2_x, temp_segment0,0)
  assert(ntriangles < max_triangles)
  nt = 3 // Number of points in a triangle !
  do j = 0, ntriangles - 1
    increment_g2_num_polygons
    do i = 0, nt

```

```

 $g2\_polygon\_xz_{g2\_num\_polygons,i,g2\_x} = temp\_triangles_{j,i,g2\_x}$ 
 $g2\_polygon\_xz_{g2\_num\_polygons,i,g2\_z} = temp\_triangles_{j,i,g2\_z}$ 
 $g2\_polygon\_segment_{g2\_num\_polygons,i} = temp\_segment_{j,i}$  /* Transfer data on auxiliary strata
to main arrays. This is more complicated than breakup_polygon case above since we need
to search the triangles to identify which one contains the desired segment. */
if((temp_num_stratum_pts > 0)  $\wedge$  (i  $\neq$  3)) then
  do k = 1, temp_num_stratum_pts
    /* The following is a little subtle. The user needs to specify polygons in the clockwise
     direction in order to preserve segment number. First, assume that the user does
     likewise in specifying auxiliary strata. Then, the selected polygon segment consists
     of the point indicated by temp_stratum_ptsk and that point number plus one. The
     corresponding triangle side must be labeled with both of these point numbers.
     Again, because we know the triangles are all clockwise, we can assume that the i + 1
     segment would match the larger point number. Furthermore, since the i = 0 and
     i = 3 points are the same, we don't have to worry that i = 3 will be hit first. One
     thing missed on first implementation was the need for the mod function here when
     the desired segment is the last one in the original polygon so that the second point in
     the segment has been assigned 0 instead of n. Also, initial code just had the assert
     below on i, allowing temp_segmentj,4 to be evaluated, accessing undefined memory.
     Instead, we now exclude i = 3 from the above if-then clause. */
    if((temp_segmentj,i  $\equiv$  temp_stratum_ptsk)  $\wedge$  (temp_segmentj,i+1  $\equiv$ 
      mod(temp_stratum_ptsk + 1, n))) then
      assert(i  $\neq$  3)
      increment_num_aux_sectors
      aux_stratum_num_aux_sectors = temp_aux_stratum
      aux_stratum_poly_num_aux_sectors = g2_num_polygons
      aux_stratum_points_num_aux_sectors = i /* As above, the segment number for a given
         stratum is just an enumeration of the segments comprising the stratum. */
      temp_aux_segment = 1
      if(num_aux_sectors > 1) then
        do i_aux = num_aux_sectors - 1, 1, -1
          if(aux_stratumi_aux  $\equiv$  temp_aux_stratum) then
            temp_aux_segment = aux_stratum_segmenti_aux + 1
            go to sector_break2
          end if
        end do
      sector_break2: continue
      end if
      aux_stratum_segment_num_aux_sectors = temp_aux_segment
    end if
    end do
  end if
  end do
  end do
  g2_polygon_pointsg2_num_polygons = nt
  if(process_polygon  $\equiv$  triangulate_to_zones  $\wedge$  j > 0)
    zone = zone + 1 /* Write out stratum numbers with each of the newly created zones.
       Should probably incorporate this information into geometry2d. */
  @#if 0
    if(process_polygon  $\equiv$  triangulate_to_zones  $\vee$  j  $\equiv$  0) then
      write(35, *) zone, current_int_props poly_stratum
    end if
  @#endif

```

```

g2_polygon_zoneg2_num_polygons = zone
g2_polygon_stratumg2_num_polygons = current_int_props poly_stratum
do i_prop = 1, poly_int_max
    poly_int_propsg2_num_polygons,i_prop = current_int_propsi_prop
end do
do i_prop = 1, poly_real_max
    poly_real_propsg2_num_polygons,i_prop = current_real_propsi_prop
end do
@if 0
if ((new_zone_type ≡ "solid") ∨ (new_zone_type ≡ "exit") ∨ (process_polygon ≡
triangulate_polygon)) then
@else
/* Have removed consideration of zone type in deciding whether or not to resolve zones in
the y direction. This is now controlled entirely by process_polygon. */
if (process_polygon ≡ triangulate_polygon) then
#endiff
solid_zone0 = zone
one_zone_type0 = new_zone_type
call decompose_polygon(nt, g2_polygon_xzg2_num_polygons,0,g2_x, solid_zone, 1, solid_faces)
else
@if 0
assert((new_zone_type ≡ "plasma") ∨ (new_zone_type ≡ "vacuum") ∨ (new_zone_type ≡
"mixed"))
#endiff
assert(process_polygon ≡ triangulate_to_zones)
do i = 0, y_div - 1
    if (i > 0)
        zone = zone + 1
    zonearrayi = zone
    if (new_zone_type ≠ "mixed")
        zone_type_arrayi = new_zone_type
    end do
    call decompose_polygon(nt, g2_polygon_xzg2_num_polygons,0,g2_x, zonearray, y_div, facearray)
end if
if (j ≡ 0 ∨ process_polygon ≡ triangulate_to_zones)
    call triangle_centroid(temp_trianglesj,0,g2_x, center)
#endiff
@if 0
if ((new_zone_type ≡ "solid") ∨ (new_zone_type ≡ "exit") ∨ (process_polygon ≡
triangulate_polygon)) then
@else
if (process_polygon ≡ triangulate_polygon) then
#endiff
call update_zone_info(solid_zone, ix, iz, one_zone_type, nt,
g2_polygon_xzg2_num_polygons,0,g2_x, center, 1, solid_ys)
else
call update_zone_info(zonearray, ix, iz, zone_type_array, nt,
g2_polygon_xzg2_num_polygons,0,g2_x, center, y_div, y_values)
end if
end do
end if
end if

```

This code is used in section 9.

## 15 Set up trivial end zones

These are needed for “hardware” (quasi-3D) symmetry cases. The material is specified above and set below with the corresponding sectors. Do this at the end just in case having all of the plasma and vacuum zones defined first turns out to be useful.

```

⟨ Setup End Zones 15 ⟩ ≡
  if (symmetry ≡ geometry-symmetry-plane-hw) then
    vc_set(a_y, zero, one, zero)
    call plane(min_corner, a_y, coeff)
    ix = 0
    iz = 0
    end_faces0 = define_surface(coeff, T)
    end_faces1 = solid_faces0
    zone++
    n = 0
    poly4n,g2_x = min_corner1
    poly4n,g2_z = min_corner3
    n++
    poly4n,g2_x = min_corner1
    poly4n,g2_z = max_corner3
    n++
    poly4n,g2_x = max_corner1
    poly4n,g2_z = max_corner3
    n++
    poly4n,g2_x = max_corner1
    poly4n,g2_z = min_corner3
    n++
    poly4n,g2_x = poly40,g2_x
    poly4n,g2_z = poly40,g2_z
    solid_zone0 = zone
    end_zones0 = zone
    call decompose_polygon(n, poly40,g2_x, solid_zone, 1, end_faces)
    call quad_center(poly40,g2_x, center)
    one_zone_type0 = "solid"
    solid_ys0 = min_corner2
    solid_ys1 = y_values0
    call update_zone_info(solid_zone, ix, iz, one_zone_type, n, poly40,g2_x, center, 1, solid_ys)
      /* Back side */
    end_faces0 = solid_faces1
    call plane(max_corner, a_y, coeff)
    end_faces1 = define_surface(coeff, T)
    zone++
    solid_zone0 = zone
    end_zones1 = zone
    call decompose_polygon(n, poly40,g2_x, solid_zone, 1, end_faces)
    one_zone_type0 = "solid"
    solid_ys0 = y_valuesy_div
    solid_ys1 = max_corner2
    call update_zone_info(solid_zone, ix, iz, one_zone_type, n, poly40,g2_x, center, 1, solid_ys)
      /* Cylindrical section, front side */
  else if (symmetry ≡ geometry-symmetry-cyl_section) then

```

```

cos_y = cos(min_corner_2)
sin_y = sin(min_corner_2)
vc_set(a_y, -sin_y, cos_y, zero)
vc_set(b_y, max_corner_1 * cos_y, max_corner_1 * sin_y, max_corner_3)
call plane(b_y, a_y, coeff)
ix = 0
iz = 0
end_faces_0 = define_surface(coeff, T)
end_faces_1 = solid_faces_0
zone++
n = 0
poly4_n,g2_x = min_corner_1
poly4_n,g2_z = min_corner_3
n++
poly4_n,g2_x = min_corner_1
poly4_n,g2_z = max_corner_3
n++
poly4_n,g2_x = max_corner_1
poly4_n,g2_z = max_corner_3
n++
poly4_n,g2_x = max_corner_1
poly4_n,g2_z = min_corner_3
n++
poly4_n,g2_x = poly4_0,g2_x
poly4_n,g2_z = poly4_0,g2_z
solid_zone_0 = zone
end_zones_0 = zone
call decompose_polygon(n, poly4_0,g2_x, solid_zone, 1, end_faces)

call quad_center(poly4_0,g2_x, center)
one_zone_type_0 = "solid"
solid_ys_0 = min_corner_2
solid_ys_1 = y_values_0
call update_zone_info(solid_zone, ix, iz, one_zone_type, n, poly4_0,g2_x, center, 1, solid_ys)
    /* Back side. As was done in setting up the universal cell surfaces, need to look out for the
       special case of an exactly 180 degree section. */
if (abs(max_corner_2 - min_corner_2 - PI) > epsilon_angle) then
    cos_y = cos(max_corner_2)
    sin_y = sin(max_corner_2)
    vc_set(a_y, -sin_y, cos_y, zero)
    vc_set(b_y, max_corner_1 * cos_y, max_corner_1 * sin_y, max_corner_3)
    call plane(b_y, a_y, coeff)
    end_faces_1 = define_surface(coeff, T)
else /* The bounding faces are identical; need the minus sign to counter the one appearing in
       process_polygon_cylindrical. */
    end_faces_1 = -end_faces_0
end if
end_faces_0 = solid_faces_1
zone++
solid_zone_0 = zone
end_zones_1 = zone
call decompose_polygon(n, poly4_0,g2_x, solid_zone, 1, end_faces)
one_zone_type_0 = "solid"

```

```
solid_ys0 = y-valuesy-div
solid_ys1 = max-corner2
call update_zone_info(solid_zone, ix, iz, one_zone_type, n, poly40,g2-x, center, 1, solid_ys)
end if
```

This code is used in section 9.

## 16 Set up toroidally facing sectors

These are needed for toroidally mixed zone types in 3-D cases.

```

⟨ Setup Toroidally Facing Sectors 16 ⟩ ≡
  if ((symmetry ≡ geometry-symmetry-cyl-hw) ∨ (symmetry ≡
    geometry-symmetry-cyl-section) ∨ (symmetry ≡ geometry-symmetry-plane-hw))
    then
      assert(y-div > 0)
      if (symmetry ≡ geometry-symmetry-cyl-hw) then
        y-max = y-div - 1 // First and last faces identical
      else
        y-max = y-div
      end if
      solid-sector = int-undef
      other-sector = int-undef
      do i = 0, y-max /* Don't need to repeat for other side since it checks both sides. Should get
        num-zone1 > 1 and num-zone2 > 1 in the typical case. The principal exception should be
        the end zones in a box or toroidal section case. */
        face1 = abs(facearrayi)
        call find-poly-zone(face1, zn-undefined, zn-undefined, g2-num-polygons, sect-zone1,
          num-zone1, sect-zone2, num-zone2)
        if ((num-zone1 > 0) ∧ (num-zone2 > 0)) then
          other-seg = 0
          do k-zone1 = 1, num-zone1
            do k-zone2 = 1, num-zone2
              assert((num-zone1 ≡ num-zone2) ∨ (num-zone1 ≡ 1) ∨ (num-zone2 ≡ 1))
              if ((num-zone1 ≡ 1) ∨ (num-zone2 ≡ 1) ∨ (zn-index(sect-zone2k-zone2,
                zi_ptr) ≡ zn-index(sect-zone1k-zone1,
                zi_ptr))) ∧ (zn-type(sect-zone1k-zone1) ≠ zn-type(sect-zone2k-zone2)) ∧
                (zn-type(sect-zone1k-zone1) ≠ zn-exit) ∧ (zn-type(sect-zone2k-zone2) ≠ zn-exit))
                then
                  if (zn-type(sect-zone1k-zone1) ≡ zn-solid) then
                    solid-zone-p = sect-zone1k-zone1 // solid-zone used as array elsewhere
                    solid-face = face1
                    other-zone = sect-zone2k-zone2
                    other-face = -face1
                  else if (zn-type(sect-zone2k-zone2) ≡ zn-solid) then
                    solid-zone-p = sect-zone2k-zone2
                    solid-face = -face1
                    other-zone = sect-zone1k-zone1
                    other-face = face1 /* No other else since we can certainly have matching zone types
                      adjacent to one another. */
                  end if /* For the end zones, we need to find their stratum numbers, etc. from the
                    corresponding arrays. For everything else, can search for the matching polygon.
                    If y-p-stratum is set in the input file, sectors are defined on the adjacent plasma
                    or vacuum zones. Note that the value of other-stratum is tested below against
                    int-unused to enforce this. The segment number is incremented for these sectors to
                    provide a distinguishing label in case they are used to define a diagnostic. */
                  if (((symmetry ≡ geometry-symmetry-plane-hw) ∨ (symmetry ≡
                    geometry-symmetry-cyl-section)) ∧ ((solid-zone-p ≡ end-zones0) ∨ (solid-zone-p ≡
                    end-zones1))) then

```

```

if (solid_zone_p  $\equiv$  end_zones0) then
    this_stratum = y_stratum0
    this_mat = y_mat0
    other_stratum = y_p-stratum0
else
    this_stratum = y_stratum1
    this_mat = y_mat1
    other_stratum = y_p-stratum1
end if
this_seg = 1
other_seg++
this_temp = const(3., 2) * boltzmanns_const
this_rc = one
else /* Search the polygon information for the stratum number to be used. In this
       case, the “other” stratum and segment numbers are just set equal to the values
       found here for simplicity. */
this_poly = int_undef
do i_poly = 1, g2_num_polygons
    if (g2_polygon_zonei_poly  $\equiv$  zn_index(solid_zone_p, zi_ptr)) then
        assert(this_poly  $\equiv$  int_undef)
        this_poly = i_poly
    end if
end do
assert(this_poly  $\neq$  int_undef)
this_stratum = poly_int_propsthis_poly.poly_stratum
other_stratum = this_stratum
this_mat = poly_int_propsthis_poly.poly_material
this_temp = poly_real_propsthis_poly.poly_temperature * boltzmanns_const
this_rc = poly_real_propsthis_poly.poly_recyc_coeff /* Set the segment number to one
               larger than the largest one in use for this stratum. Note that this requires that
               setup_sectors be called before this since it uses the segment labels 1 through the
               number of polygon sides. */
this_seg = int_undef
do i_sect = 1, nsectors
    if (stratai_sect  $\equiv$  poly_int_propsthis_poly.poly_stratum) then
        if (this_seg  $\equiv$  int_undef) then
            this_seg = sector_strata_segmenti_sect
        else
            this_seg = max(this_seg, sector_strata_segmenti_sect)
        end if
    end if
end do
assert((this_seg  $\geq$  0)  $\wedge$  (this_seg  $\neq$  int_undef)) // Temporary diagnostic
this_seg++
other_seg = this_seg
end if
/* To prevent redundant sector definitions (e.g., for num_zone1 = 1, num_zone2 > 1),
   check to see if the sector parameters compiled on this pass through the loop match
   the ones last used to define the sector of this type. */
new_solid_sector = TRUE
if (solid_sector  $\neq$  int_undef) then

```

```

    if (sc_compare(solid_sector, solid_zone_p, solid_face, this_stratum, this_seg))
        new_solid_sector = FALSE
    end if
    new_other_sector = TRUE
    if (other_sector ≠ int_undef) then
        if (sc_compare(other_sector, other_zone, other_face, other_stratum, other_seg))
            new_other_sector = FALSE
    end if /* Define new sectors as needed. */
    if (new_solid_sector ≡ TRUE) then
        solid_sector = define_sector(this_stratum, this_seg, solid_face, solid_zone_p, other_zone)
        other_type = zn_type(other_zone)
        if (other_type ≡ zn_plasma) then
            define_sector_target(solid_sector, this_mat, this_temp, this_rc)
        else if (other_type ≡ zn_vacuum) then
            define_sector_wall(solid_sector, this_mat, this_temp, this_rc)
        else
            assert('Unexpected_zone_type' ≡ ' ')
        end if
    end if
    if ((new_other_sector ≡ TRUE) ∧ (other_stratum ≠ int_unused)) then
        other_sector = define_sector(other_stratum, other_seg, other_face, other_zone,
                                      solid_zone_p)
        other_type = zn_type(other_zone)
        if (other_type ≡ zn_plasma) then
            define_sector_plasma(other_sector)
        else if (other_type ≡ zn_vacuum) then
            define_sector_vacuum(other_sector)
        else
            assert('Unexpected_zone_type' ≡ ' ')
        end if
    end if
    end if // adjacent zone types differ
end do // k_zone2
end do // k_zone1
end if // Nonzero num_zone1 and num_zone2
end do // Loop over y index
end if // Toroidally segmented symmetry

```

This code is used in section 9.

## 17 Set up sector based diagnostics, including those composed of auxiliary strata

```

⟨Setup Sector Diagnostics 17⟩ ≡
if (num_diags > 0) then
  do i_diag = 1, num_diags
    i_grp = 0
    do i_sect = 1, nsectors
      if (stratai_sect ≡ diag_stratumi_diag) then /* The auxiliary sectors were set up in such a way
          that the above test identifies them adequately. So, if this sector is an auxiliary sector, we
          can add it to the list for this diagnostic without any further checking. */
        is_aux_sector = FALSE
      do i_aux = 1, num_aux_sectors
        if ((stratai_sect ≡ aux_stratumi_aux) ∧ (sector_strata_segmenti_sect ≡
          aux_stratum_segmenti_aux)) then
          is_aux_sector = TRUE
        end if
      end do
      if (is_aux_sector ≡ TRUE) then
        i_grp++
        assert(i_grp ≤ max_grp_sectors) // Dim. of grp_sectors
        grp_sectorsi_grp = i_sect
      else /* However, if the diag_stratum in question is associated with a default sector, we
          need to dig further since the sectors on both side of a surface are assigned the same
          stratum number. We then use the polygon information to select only the sectors on
          one side of (both sides use the same stratum number). If we allow sectors on both
          sides in a diagnostics, the directional independent variables (mass_in, etc.) will not
          work properly. */
        if (((diag_solidi_diag ≡ TRUE) ∧ (sc_target_check(sector_type_pointeri_sect, sc_target) ∨
          sc_wall_check(sector_type_pointeri_sect, sc_wall) ∨
          sc_exit_check(sector_type_pointeri_sect, sc_exit))) ∨ ((diag_solidi_diag ≡
          FALSE) ∧ (sc_plasma_check(sector_type_pointeri_sect, sc_plasma) ∨
          sc_vacuum_check(sector_type_pointeri_sect, sc_vacuum))) then
          i_grp++
          assert(i_grp ≤ max_grp_sectors) // Dim. of grp_sectors
          grp_sectorsi_grp = i_sect
        end if
      end if
    end if
  end do
@#if 0
  do i_poly = 1, g2_num_polygons
    if ((g2_polygon_stratumi_poly ≡ diag_stratumi_diag) ∧ (zn_index(g2_polygon_zonei_poly,
      zi_ptr) ≡ zn_index(sector_zonei_sect, zi_ptr)) then
      i_grp++
      assert(i_grp ≤ max_grp_sectors) // Dim. of grp_sectors
      grp_sectorsi_grp = i_sect
    end if
  end do
end if
end if

```

```
    end do
@#endiff
    assert( $i\_grp > 0$ )
    assert( $diag\_name_{i\_diag} \neq char\_uninit$ )
    call diag_grp_init( $diag\_name_{i\_diag}$ ,  $i\_grp$ ,  $diag\_variable_{i\_diag}$ ,  $diag\_tab\_index_{i\_diag}$ ,  $diag\_var\_min_{i\_diag}$ ,
                       $diag\_var\_max_{i\_diag}$ ,  $diag\_mult_{i\_diag}$ ,  $diag\_spacing_{i\_diag}$ ,  $grp\_sectors$ )
end do
end if
```

This code is used in section 9.

## 18 Read mesh file from UEDGE

```
"definegeometry2d.f" 18 ≡
@m max_xpts 2

⟨Functions and subroutines 8⟩ +≡
subroutine read_uedge_mesh(nunit, nxd, nzd, nxpt, mesh_xz)

implicit none_f77
implicit none_f90

integer nunit, nxd, nzd, nxpt // Input
real mesh_xznzd,nxd,0:4,g2_x:g2_z // Output
integer ix, iz, i // Local
integer corner_ptr0:4, iysptrx1max_xpts, iysptrx2max_xpts, ixlbmax_xpts, ixpt1max_xpts,
      ixmdpmax_xpts, ixpt2max_xpts, ixrbmax_xpts
real dummy /* This file format now deals with both single and double nulls. Although we don't
   really need to know which for present purposes (since we have corner data for all zones), we
   do need to use the number of X-points, nxpt, to count off the initial lines. All of the other
   indices are read in for completeness and possible future use. */

assert((nxpt ≥ 1) ∧ (nxpt ≤ max_xpts))
do i = 1, nxpt
  read(nunit, *) iysptrx1i, iysptrx2i
  read(nunit, *) ixlbi, ixpt1i, ixmdpi, ixpt2i, ixrbi
end do /* The specific ordering to be used in the rest of this code is that a polygon will be traced
   out in the clockwise direction by following the corners in order 1 → 4. The center is denoted as
   corner 0. The corner_ptr array is used to translate the known orientation of UEDGE mesh
   corners into the desired one. */
corner_ptr0 = 0
corner_ptr1 = 1
corner_ptr2 = 4
corner_ptr3 = 2
corner_ptr4 = 3
read(nunit, *) SP(((mesh_xziz,ix,corner_ptri,g2_x, ix = 1, nxd), iz = 1, nzd), i = 0, 4)
read(nunit, *) SP(((mesh_xziz,ix,corner_ptri,g2_z, ix = 1, nxd), iz = 1, nzd), i = 0, 4)
dummy = zero
open(unit = nunit + 1, file = 'uedge_sonnet', status = 'unknown', form = 'formatted')
write(nunit + 1, *)
write(nunit + 1, *) 'Element output:'
write(nunit + 1, *)
write(nunit + 1, '(a,f16.13)') 'R*Btor=0', dummy
write(nunit + 1, *) 'ncut=0'
write(nunit + 1, *)
do iz = 1, nzd
  do ix = 1, nxd /* The specific ordering to be used in the rest of this code is that a polygon will
    be traced out in the clockwise direction by following the corners in order 1 → 4. The corners
    used here with mesh_x and mesh_z are chosen so as to translate the known orientation of
    the Sonnet mesh corners into the desired one. The center is denoted as corner 0. */
  write(nunit + 1, '(a,i4,a,i3,a,i3,a,e17.10,a,e17.10,a,6x,a,e17.10,a,e17.10,a)')
    'Element', (iz - 1) * nxd + ix - 1, '(', ix - 1, ',', ',', iz - 1, ')':(',
      mesh_xziz,ix,2,g2_x, ', ', mesh_xziz,ix,2,g2_z, ')', ',',
      mesh_xziz,ix,3,g2_x, ', ', ',', mesh_xziz,ix,3,g2_z, ')'
```

```

write(nunit + 1, '(a,e17.10,13x,a,e17.10,a,e17.10,a)')'uuuField_ratiouu='dummy,
      '(', mesh_xz_iz,ix,0,g2_x, ',', mesh_xz_iz,ix,0,g2_z, ')'
write(nunit + 1, '(29x,a,e17.10,a,e17.10,a,6x,a,e17.10,a,e17.10,a)')'(',
      mesh_xz_iz,ix,1,g2_x, ',', mesh_xz_iz,ix,1,g2_z, ')', '(', mesh_xz_iz,ix,4,g2_x, ',', 
      mesh_xz_iz,ix,4,g2_z, ')
write(nunit + 1, *)'uu-----\\
-----,
end do
end do
close(unit = nunit + 1)

return
end

```

## 19 Compute minimum and maximum coordinates

Search *nodes* array for maximum and minimum coordinate values that have been entered so far. These can then be used in setting the arguments of the *bounds* keyword.

```

⟨ Functions and subroutines 8 ⟩ +≡
subroutine find_min_max(num_nodes, nodes, node_type, x_min, x_max, z_min, z_max)
  implicit none_f77
  implicit none_f90

  integer num_nodes // Input
  integer node_type*
  real nodes*,g2_x:g2_z

  real x_min, x_max, z_min, z_max // Output
  integer inode // Local
  x_min = const(1., 5)
  x_max = zero
  z_min = const(1., 5)
  z_max = zero

  do inode = 1, num_nodes
    if (node_typeinode ≠ node_no_elements) then
      x_min = min(x_min, nodesinode,g2_x)
      x_max = max(x_max, nodesinode,g2_x)
      z_min = min(z_min, nodesinode,g2_z)
      z_max = max(z_max, nodesinode,g2_z)
    end if
  end do

  assert(x_max > x_min)
  assert(z_max > z_min)

  return
end

```

## 20 Print wall coordinates to a file for user examination

There are two possible formats. The first (“tabular”) has the walls laid out in successive columns, suitable for plotting with an external program. The second (“linear”) is consistent with the format used by the “wallfile” keyword. The user can provide the name of the file. If absent, the data are written to stdout.

```
"definegeometry2d.f" 20 ≡
@m table_size 4 // Walls per table section

⟨Functions and subroutines 8⟩ +≡
subroutine print_walls(nunit, file_format, walloutfile, num_walls, wall_segment_count, wall_nodes,
nodes)
implicit none_f77
implicit none_f90

integer nunit, num_walls // Input
integer wall_segment_count_*, wall_nodes_*,0:g2_num_points-1
real nodes_*,g2_x:g2_z
character*LINELEN file_format
character*FILELEN walloutfile

integer num_sections, isec, istart, iend, max_segs, /* Local */
iwall, iseg, inode, b, e
character*csec
character*130 wall_line // Holds table_size (i.e., 4) pairs
character*31 wall_chunk // Space to write x,z pair

st_decls

if (file_format == 'linear') then
  if (walloutfile != char_undef) then
    open(unit = nunit, file = trim(walloutfile), status = 'unknown')
  end if
  write(nunit, *) num_walls
  write(nunit, *) SP(wall_segment_count_iwall + 1, iwall = 1, num_walls)
  do iwall = 1, num_walls
    do iseg = 0, wall_segment_count_iwall
      inode = wall_nodes_iwall,iseg
      write(nunit, '(1x,1pe18.10,2x,e18.10)') nodes_inode,g2_x, nodes_inode,g2_z
    end do
  end do
else if (file_format == 'tabular') then
  num_sections = (num_walls / table_size)
  if (num_sections * table_size < num_walls) then
    num_sections++
  else if (num_sections * table_size > num_walls) then
    assert('Problem calculating num_sections' == ' ')
  end if
  do isec = 1, num_sections
    if (walloutfile != char_undef) then
      write(csec, '(i2.2)') isec
      open(unit = nunit, file = trim(walloutfile) || csec, status = 'unknown')
    end if
  end do
end if
```

```

isstart = table_size * (isec - 1) + 1
iend = min(table_size * isec, num_walls)
max_segs = 0
do iwall = isstart, iend
    max_segs = max(max_segs, wall_segment_countiwall)
end do
do iseg = -1, max_segs
    wall_line = '||'
    b = 1
    e = 2
    do iwall = isstart, iend
        if (iseg ≡ -1) then
            write(wall_chunk, '(5x,a,i2.2,11x,a,i2.2,6x)') 'X_', iwall, 'Z_', iwall
        else if (iseg ≤ wall_segment_countiwall) then
            inode = wall_nodesiwall,iseg
            write(wall_chunk, '(1pe14.6,1x,e14.6,2x)') nodesinode,g2_x, nodesinode,g2_z
        else /* KaleidaGraph likes using periods to denote empty data cells. It appears that
               Excel can live with them as well. */
            write(wall_chunk, '(7x,a,14x,a,7x)') '.', '.'
        end if
        wall_line = wall_line(b : e) || wall_chunk
        e += 30
    end do
    write(nunit, *) wall_line(b : e)
end do
if (walloutfile ≡ char_undef) then
    write(nunit, *)
    write(nunit, *) '-----',
    write(nunit, *)
else
    close(unit = diskout)
end if
end do
else
    write(stderr, *) 'The format for the wall file must be either "linear" or "tabular"',
end if

return
end

```

## 21 Read input lines specifying a polygon

The structures in the argument list can get a little confusing. Let's spell them out here at least.

*nunit* The logical unit number to which the input file is attached.

*num\_walls* Number of walls created during the preparatory stage, either from a *wallfile* or a *dg\_file* command.

*wall\_segment\_count* Number of elements (or, equivalently, segments) in each of these walls.

*wall\_nodes* List of nodes comprising these walls. Because the wall nodes and elements are constructed together at the same time, they are trivially related and only the nodes are needed here. This array just contains pointers into the *nodes* array.

*nodes* Coordinates of all of the nodes specified in the preparatory stage via the *wallfile* and *dg\_file* commands.

*dg\_elements\_list* List of the “start” and “end” nodes associated with each element described in the *dg\_file*.

*num\_dg\_poly* Number of “polygons” (in the DG sense) contained in the *dg\_file*. Note that these may be open or closed.

*dg\_poly\_num\_elements* Number of elements in each of the DG polygons contained in the *dg\_file*.

*dg\_polygons* List of elements in each of the DG polygons contained in the *dg\_file*.

*dg\_poly\_num\_meshcon* Number of mesh connections (referred to as “meshcon”) associated with each DG polygon. Currently, the only valid values are 0, 1, or 2.

*dg\_poly\_meshcon* Integer label partially identifying the mesh connections associated with each DG polygon.

*dg\_poly\_meshcon\_hv* Switch ( $= mesh\_h \leftrightarrow$  “horizontal” or  $= mesh\_v \leftrightarrow$  “vertical”) completing the identification of the mesh connections associated with each DG polygon.

*xb\_min* Minimum value of *x* as set with the *bounds* command.

*xb\_max* Maximum value of *x* as set with the *bounds* command.

*zb\_min* Minimum value of *z* as set with the *bounds* command.

*zb\_max* Maximum value of *z* as set with the *bounds* command.

*nxd* Number of cells in the first dimension of the input Sonnet or UEDGE mesh.

*nzd* Number of cells in the second dimension of the input Sonnet or UEDGE mesh.

*mesh\_xz* Coordinates of the input Sonnet or UEDGE mesh.

*mesh\_nodes* A list of (unique) nodes created from the four edges of the input Sonnet or UEDGE mesh. This array contains the coordinates rather than pointers to another array. These nodes are independent of those mentioned above.

*mesh\_elements* Full list of the “start” and “end” nodes associated with each element created from the four edges of the input Sonnet or UEDGE mesh. No assumptions are made regarding the order or connectivity of the entries in this list.

*mesh\_edge\_num\_elements* Number of elements (or segments) on each of the four edges of the input Sonnet or UEDGE mesh.

*mesh\_edge\_elements* List of (with *mesh\_edge\_num\_elements* telling how many) elements in each of the four edges of the input Sonnet or UEDGE mesh.

*mesh\_edge\_dg\_label* Integer used by DG to partially label mesh elements.

*mesh\_edge\_hv* Switch ( $= mesh\_h \leftrightarrow \text{“horizontal”}$  or  $mesh\_v \leftrightarrow \text{“vertical”}$ ) that completes DG’s label for the mesh elements.

*mesh\_curve\_num* Array dimensioned in the calling program for use below by the *iedge* command. It will contain labels identifying which of one or more simply connected “curves” (equivalent to “walls”) created from the mesh edges.

*mesh\_scratch* Array dimensioned in the calling program to provide scratch space for the *iedge* command.

```

"definegeometry2d.f" 21 ==
@m dim_curves 5 // Used in determining topology of mesh edges
@m curve_open 1
@m curve_closed 2

@m order_undefined 0 // Used to keep track of order of iedge edges
@m order_decreasing -1
@m order_increasing 1

@m poly_loop #:0
@m outer_loop #:0
@m iedge_loop #:0
@m forward_loop #:0
@m backward_loop #:0

⟨ Functions and subroutines 8 ⟩ +≡
subroutine specify_polygon(nunit, num_walls, wall_segment_count, wall_nodes, nodes,
    dg_elements_list, num_dg_poly, dg_poly_num_elements, dg_polygons, dg_poly_num_meshcon,
    dg_poly_meshcon, dg_poly_meshcon_hv, xb_min, xb_max, zb_min, zb_max, nxd, nzd, mesh_xz,
    mesh_nodes, mesh_elements, mesh_edge_num_elements, mesh_edge_elements,
    mesh_edge_dg_label, mesh_edge_hv, mesh_curve_num, mesh_scratch, polygon, n, int_props,
    real_props, aux_stratum, aux_stratum_pts, num_aux_stratum_pts, process_polygon)

implicit none f77
ma_common // Common
implicit none f90

integer nunit, num_walls, num_dg_poly, nxd, nzd // Input
integer wall_segment_count_*, wall_nodes_*, dg_elements_list_*, element_start:element_end,
    dg_poly_num_elements_*, dg_polygons_*, dg_poly_num_meshcon_*,,
    dg_poly_meshcon_*:meshcon_max,1:meshcon_elem_max,
    dg_poly_meshcon_hv_*:meshcon_max,1:meshcon_elem_max,
    mesh_elements_*,element_start:element_end, mesh_edge_num_elements mesh_num_edges,
    mesh_edge_elements_*,mesh_num_edges, mesh_edge_dg_label_*,mesh_num_edges,
    mesh_edge_hv mesh_num_edges, mesh_curve_num_*, mesh_scratch_*
real xb_min, xb_max, zb_min, zb_max
real nodes_*,g2_x:g2_z, mesh_xz_nzd,nxd,0:4,g2_x:g2_z, mesh_nodes_*,g2_x:g2_z

integer n, process_polygon, aux_stratum, /* Output */
    num_aux_stratum_pts
integer int_props_1:poly_int_max, aux_stratum_pts_0:g2_num_points-1
real polygon_0:g2_num_points-1,g2_x:g2_z, real_props_1:poly_real_max

integer length, p, b, e, i, wall, start, stop , temp, /* Local*/
    ix_start, ix_stop, iz_start, iz_stop, ix_step, iz_step, corner, ixi, izi, num, ix, iz, nout, step, i_prop,
    aux_stratum_def, aux_stratum_start, poly, j, num_ends, end_point, end_tip, print_polygon,
    iedge_points, num_curves, num_iedges, i_tot, closed_curve, j_min, j_max, n_iedge, iedge_order,
    i_mc, swap_edges, tmp_edge, tmp_seg, j_init, j_fin, common_node, edge_reverse, xcut
integer end_nodes_2*g2_num_points, end_elements_2*g2_num_points, end_tips_2*g2_num_points,
    curve_type dim_curves, curve_start dim_curves, curve_start_tip dim_curves,
    iedges mesh_num_edges, mc_edge_1:meshcon_max,1:meshcon_elem_max,
    mc_seg_1:meshcon_max,1:meshcon_elem_max, mc_node_4
real poly_area, min_dist
real iedge_temp_0:g2_num_points-1,g2_x:g2_z, dist_4
character*FILELEN polyoutfile

```

```

character*LINELEN line, keyword

vc_decl(poly-p2)
vc_decl(iedge-p2)
vc_decl(delta)

st_decls
vc_decls

external polygon_area
real polygon_area

process_polygon = int_undef
print_polygon = FALSE
n = 0
poly = int_undef
iedge_points = 0 // Need this still?
num_edges = 0 /* Set these to peculiar values. If not explicitly set in this routine, they will be
                  overridden by the "current" values in the calling routine. */
do i_prop = 1, poly_int_max
    int_propsi_prop = int_uninit
end do
do i_prop = 1, poly_real_max
    real_propsi_prop = real_uninit
end do

aux_stratum = int_uninit
num_aux_stratum_pts = 0
aux_stratum_def = FALSE

poly-loop: continue
    assert(read_string(diskin, line, length))
    assert(length ≤ len(line))
    length = parse_string(line(: length))
    p = 0
    assert(next_token(line, b, e, p))
    keyword = line(b : e)

    if (keyword ≡ 'outer') then
        ⟨ Outer Keyword 21.1 ⟩
    else if (keyword ≡ 'wall') then
        ⟨ Wall Keyword 21.2 ⟩
    else if (keyword ≡ 'edge') then
        ⟨ Edge Keyword 21.3 ⟩
    else if (keyword ≡ 'dg_polygon') then
        ⟨ DG Polygon Keyword 21.4 ⟩
    else if (keyword ≡ 'iedge') then
        ⟨ Intelligent Edge Specification 21.5 ⟩
    else if (keyword ≡ 'reverse') then
        assert(n > 0)
        /* Use n - 1 here since the last point doesn't get set until process_polygon is set. */
        call reverse_polygon(n - 1, polygon0,g2_x)
    else if (keyword ≡ 'stratum') then

```

```

assert(next_token(line, b, e, p))
int_props_poly_stratum = read_integer(line(b : e))

else if (keyword ≡ 'material') then
  assert(next_token(line, b, e, p))
  int_props_poly_material = ma_lookup(line(b : e))
  assert(ma_check(int_props_poly_material))

else if (keyword ≡ 'temperature') then
  assert(next_token(line, b, e, p))
  real_props_poly_temperature = read_real(line(b : e))
  assert(real_props_poly_temperature > zero)

else if (keyword ≡ 'recyc_coef') then
  assert(next_token(line, b, e, p))
  real_props_poly_recyc_coef = read_real(line(b : e))
  assert(real_props_poly_recyc_coef ≥ zero ∧ real_props_poly_recyc_coef ≤ one)

else if (keyword ≡ 'triangle_area') then
  assert(next_token(line, b, e, p))
  real_props_poly_min_area = read_real(line(b : e))
  assert(real_props_poly_min_area > zero)

else if (keyword ≡ 'triangle_hole') then
  assert(next_token(line, b, e, p))
  int_props_poly_num_holes = read_integer(line(b : e))
  if (int_props_poly_num_holes > 0) then
    assert(int_props_poly_num_holes ≡ 1)
    assert(next_token(line, b, e, p))
    real_props_poly_hole_x = read_real(line(b : e))
    assert(next_token(line, b, e, p))
    real_props_poly_hole_z = read_real(line(b : e))
  end if

else if (keyword ≡ 'aux_stratum') then
  /* Set up currently to handle only one auxiliary stratum per polygon. */
  assert(next_token(line, b, e, p))
  if (aux_stratum ≡ int_uninit) then
    aux_stratum = read_integer(line(b : e))
    assert(aux_stratum ≠ int_uninit)
  else
    assert(aux_stratum ≡ read_integer(line(b : e)))
  end if
  aux_stratum_def = TRUE
  aux_stratum_start = n

else if (keyword ≡ 'end_aux_stratum') then
  assert(aux_stratum_def ≡ TRUE)
  do i = aux_stratum_start, n - 1
    num_aux_stratum_pts ++
    aux_stratum_ptsnum_aux_stratum_pts = i
  end do
  assert(num_aux_stratum_pts > 0)
  aux_stratum_def = FALSE

```

```

else if (keyword ≡ 'print_polygon') then
  print_polygon = TRUE
  if (next_token(line, b, e, p)) then
    polyoutfile = line(b : e)
    nout = diskout
  else
    polyoutfile = char_undef
    nout = stdout
  end if

else if (keyword ≡ 'breakup_polygon') then
  process_polygon = breakup_polygon

else if (keyword ≡ 'triangulate_polygon') then
  process_polygon = triangulate_polygon

else if (keyword ≡ 'triangulate_to_zones') then
  process_polygon = triangulate_to_zones

else if (keyword ≡ 'clear_polygon') then
  process_polygon = clear_polygon

else
  write (stderr, *) 'Unexpected_polygon_keyword',
  process_polygon = clear_polygon
end if

if (process_polygon ≡ int_undef)
  go to poly_loop /* Everything else is done, so we can process the iedge keyword and / or mesh
                     connection data. */
⟨ Intelligent Edge Processing 21.6 ⟩
  /* Polygon is complete. Close it up if it is to be further processed. */
if (process_polygon ≠ clear_polygon) then
  polygonn,g2-x = polygon0,g2-x
  polygonn,g2-z = polygon0,g2-z
end if /* Print it out, if requested. */
if (print_polygon ≡ TRUE) then
  if (polyoutfile ≠ char_undef) then
    open (unit = nout, file = trim(polyoutfile), status = 'unknown')
  end if
  write (nout, '(8x,a,15x,a)' 'X', 'Z'
  do i = 0, n - 1
    write (nout, '(1x,1pe14.6,2x,e14.6)' polygoni,g2-x, polygoni,g2-z)
  end do
  if (polyoutfile ≠ char_undef) then
    close (unit = nout)
  end if
end if
return
end

```

Add points along universal cell to current polygon.

```

⟨ Outer Keyword 21.1 ⟩ ≡
outer_loop: continue
  if (next_token(line, b, e, p)) then
    i = read_integer(line(b : e))
    if (i ≡ 0 ∨ i ≡ 4) then
      polygonn,g2-x = xb_min
      polygonn,g2-z = zb_min
    else if (i ≡ 1) then
      polygonn,g2-x = xb_min
      polygonn,g2-z = zb_max
    else if (i ≡ 2) then
      polygonn,g2-x = xb_max
      polygonn,g2-z = zb_max
    else if (i ≡ 3) then
      polygonn,g2-x = xb_max
      polygonn,g2-z = zb_min
    end if
    n++
    assert(n ≤ g2_num_points - 1)
    go to outer_loop
  end if

```

This code is used in section 21.

Take points from a wall and add to current polygon.

```

⟨ Wall Keyword 21.2 ⟩ ≡
  assert(next_token(line, b, e, p))
  wall = read_integer(line(b : e))
  assert(wall > 0 ∧ wall ≤ num_walls)
  assert(next_token(line, b, e, p))
  if (line(b : e) ≡ '*') then
    start = 0 stop = wall_segment_countwall
  else
    start = read_integer(line(b : e))
    assert(next_token(line, b, e, p))
    if (line(b : e) ≡ '*') then stop = wall_segment_countwall
    else stop = read_integer(line(b : e))
    end if
  end if
  if (next_token(line, b, e, p)) then
    assert(line(b : e) ≡ 'reverse')
    temp = start start = stop stop = temp
  end if if ( start ≤ stop ) then
    step = 1 else if ( start > stop ) then
    step = -1
  end if
  do i = start , stop , step
    assert(i ≥ 0 ∧ i ≤ wall_segment_countwall)
    polygonn,g2-x = nodeswall-nodeswall,i:g2-x
    polygonn,g2-z = nodeswall-nodeswall,i:g2-z
    n++
    assert(n ≤ g2_num_points - 1)
  end do

```

This code is used in section 21.

Take points along a mesh edge and add to current polygon.

```

⟨Edge Keyword 21.3⟩ ≡
  assert(next_token(line, b, e, p))
  if (line(b : e) ≡ '*') then
    ix_start = 0
    ix_stop = nxd
  else
    ix_start = read_integer(line(b : e))
    assert(next_token(line, b, e, p))
    ix_stop = read_integer(line(b : e))
  end if
  assert(next_token(line, b, e, p))
  if (line(b : e) ≡ '*') then
    iz_start = 0
    iz_stop = nzd
  else
    iz_start = read_integer(line(b : e))
    assert(next_token(line, b, e, p))
    iz_stop = read_integer(line(b : e))
  end if /* The reverse and xcut options can both appear and may be in either order. Hence, the
           extended logic here. */
  edge_reverse = FALSE
  xcut = FALSE
  if (next_token(line, b, e, p)) then
    if (line(b : e) ≡ 'reverse') then
      edge_reverse = TRUE
    if (next_token(line, b, e, p)) then
      assert(line(b : e) ≡ 'xcut')
      xcut = TRUE
    end if
  else
    assert(line(b : e) ≡ 'xcut')
    xcut = TRUE
    if (next_token(line, b, e, p)) then
      assert(line(b : e) ≡ 'reverse')
      edge_reverse = TRUE
    end if
  end if
end if
if (edge_reverse ≡ TRUE) then
  temp = ix_start
  ix_start = ix_stop
  ix_stop = temp
  temp = iz_start
  iz_start = iz_stop
  iz_stop = temp
end if
if (iz_stop ≡ iz_start) then
  iz_step = 0
  if (ix_stop ≥ ix_start) then
    ix_step = 1

```

```

num = ix_stop - ix_start + 1
else
  ix_step = -1
  num = ix_start - ix_stop + 1
end if
else if (ix_stop ≡ ix_start) then
  ix_step = 0
  if (iz_stop ≥ iz_start) then
    iz_step = 1
    num = iz_stop - iz_start + 1
  else
    iz_step = -1
    num = iz_start - iz_stop + 1
  end if
end if
assert('NotFollowingAmeshEdge' ≡ ' ')
ix = ix_start
iz = iz_start
do i = 1, num
  if (xcut ≡ FALSE) then
    if (ix ≡ 0 ∧ iz ≡ 0) then
      corner = 1
      ixi = 1
      izi = 1
    else if (ix ≡ 0) then
      corner = 2
      ixi = 1
      izi = iz
    else if (iz ≡ 0) then
      corner = 4
      ixi = ix
      izi = 1
    else
      corner = 3
      ixi = ix
      izi = iz
    end if
  else // xcut == TRUE
    assert(ix ≠ 0) // A cut at ix = 0 makes no sense
    if (iz ≡ 0) then
      corner = 1
      ixi = ix
      izi = 1
    else
      corner = 2
      ixi = ix
      izi = iz
    end if
  end if
end if
polygonn,g2_x = mesh_xzizi,ixi,corner,g2_x
```

```
polygonn,g2_z = mesh_xzizi,ixi,corner,g2_z
ix += ix_step
iz += iz_step
n++
assert(n ≤ g2_num_points - 1)
end do
```

This code is used in section 21.

Process points of a polygon that was contained in the DG file. This assumes that we know nothing about the connectivity of these elements. Hence, the process is more involved than that used for creating “walls” out of DG’s elements and nodes. Note that the node characterization done at that stage is not used here. Both open and closed loops are treated.

```

⟨DG Polygon Keyword 21.4⟩ ≡
  assert(next_token(line, b, e, p))
  poly = read_integer(line(b : e))
  assert(poly > 0 ∧ poly ≤ num_dg_poly) /* The task here is to determine the desired ordering of the
                                             nodes in the DG polygon, i.e., their connectivity. In the case of an open loop, we do not know a
                                             priori which node is the first (or last) in the loop. This prevents the user from combining the
                                             dg_polygon command with any other construction keyword, except iedge. */
  assert(n ≡ 0)
  call find_endpoints(dg_elements_list, dg_poly_num_elements_poly, dg_polygons_poly, num_ends,
                       end_nodes, end_elements, end_tips)
  if (num_ends ≡ 0) then
    end_point = 1
    end_tip = element_start
  else if (num_ends ≡ 2) then
    end_point = end_elements_1
    end_tip = end_tips_1
  else /* If you see this assertion, you likely have a stray, unintended element in this dg_polygon. */
    assert('should have 0 or 2 endpoints here' ≡ ' ')
  end if
  call elements_to_polygon(nodes, dg_elements_list, dg_poly_num_elements_poly, dg_polygons_poly,
                          end_point, end_tip, n, polygon_0,g2_x)
  if (num_ends ≡ 0) then /* The polygon closes on itself. The assertion verifies this. We can then
                           check the orientation of the polygon. If counter-clockwise, reverse the order of the points. */
    assert(n ≡ dg_poly_num_elements_poly)
    assert((polygon_n,g2_x ≡ polygon_0,g2_x) ∧ (polygon_n,g2_z ≡ polygon_0,g2_z))
    poly_area = polygon_area(n, polygon_0,g2_x)
    if (poly_area < zero)
      call reverse_polygon(n, polygon_0,g2_x)
  else
    assert(n ≡ dg_poly_num_elements_poly + 1)
    /* Pretty sure this test is supposed to be the “else” version. Because the nth entry is usually
       garbage, the first version has been succeeding regardless of the situation. */
  @#if 0
    assert((polygon_n,g2_x ≠ polygon_0,g2_x) ∨ (polygon_n,g2_z ≠ polygon_0,g2_z))
  @#else
    assert((polygon_n-1,g2_x ≠ polygon_0,g2_x) ∨ (polygon_n-1,g2_z ≠ polygon_0,g2_z))
  @#endif
  end if

```

This code is used in section 21.

Isolate a mesh edge for subsequent addition to current polygon. The “intelligent” part arises from the search procedure that attempts to determine the portion of that edge the user needs. This bit of code just reads in the specification. The actual processing takes place once the *process\_polygon* command has been issued.

```

⟨Intelligent Edge Specification 21.5⟩ ≡
  assert(num_iedges ≡ 0)    // Only allow one invocation
iedge_loop: continue
  if (next_token(line, b, e, p)) then
    num_iedges++
    assert(num_iedges ≤ mesh_num_edges)
    if (line(b : e) ≡ 'S') then
      iedges_num_iedges = mesh_south
    else if (line(b : e) ≡ 'E') then
      iedges_num_iedges = mesh_east
    else if (line(b : e) ≡ 'N') then
      iedges_num_iedges = mesh_north
    else if (line(b : e) ≡ 'W') then
      iedges_num_iedges = mesh_west
    else
      write(stderr, *) 'Unexpected iedge specification,', line(b : e)
      assert(F)
    end if
    go to iedge_loop
  end if
  assert(num_iedges > 0) /* Check order and connectivity of iedges. This makes explicit use of the
                           integer values of the macros associted with these labels. More specifically, the edges are assumed
                           to trace out a counter-clockwise loop, S → E → N → W which corresponds to monotonically
                           increasing values for the integer macro values (modulo mesh_num_edges = 4). Hence, the specified
                           list must either be in this order or in the opposite of this order. In the latter case, the list is
                           reversed. */
  if (num_iedges > 1) then
    iedge_order = order_undefined
    do i = 1, num_iedges - 1
      if (mod(iedges_{i+1}, mesh_num_edges) ≡ mod((iedges_i + 1), mesh_num_edges)) then
        if (iedge_order ≡ order_undefined) then
          iedge_order = order_increasing
        else
          assert(iedge_order ≡ order_increasing)
        end if
      else if (mod(iedges_{i+1}, mesh_num_edges) ≡ mod((iedges_i - 1), mesh_num_edges)) then
        if (iedge_order ≡ order_undefined) then
          iedge_order = order_decreasing
        else
          assert(iedge_order ≡ order_decreasing)
        end if
      end if
    end do
    if (iedge_order ≡ order_decreasing) then
      do i = 1, num_iedges / 2
        j = iedges_i

```

```

 $i_{edges}_i = i_{edges}_{num\_edges-i+1}$ 
 $i_{edges}_{num\_edges-i+1} = j$ 
end do
else
    assert( $i_{edge\_order} \equiv order\_increasing$ )
end if
end if

```

This code is used in section 21.

Remainder of the code for isolating a mesh edge. This portion does the processing.

```

⟨ Intelligent Edge Processing 21.6 ⟩ ≡
if(( $n \equiv 0$ )  $\wedge$  ( $num\_edges > 0$ )) then
    ⟨ Process Closed iedge Polygon 21.7 ⟩
else if( $n > 0$ ) then
    if(( $num\_edges \equiv 0$ )  $\wedge$  ( $poly \neq int\_undef$ )) then
        if( $dg\_poly\_num\_meshcon_{poly} > 0$ ) then
            /* If you see this assertion, you may have mesh connections unintentionally set for this
               dg_polygon. Go back to DG, unmark everything and hit the “set” button for both of these
               mesh connections (see that they are listed as “Empty”). */
            assert( $dg\_poly\_num\_meshcon_{poly} \equiv 1$ )
            ⟨ Process Single Mesh Point 21.8 ⟩
        else
            assert( $dg\_poly\_num\_meshcon_{poly} \equiv 0$ ) /* Do nothing; don’t need iedge processing. */
        end if
    else if( $num\_edges > 0$ ) then
        assert( $poly \neq int\_undef$ )
        assert( $dg\_poly\_num\_meshcon_{poly} \equiv 2$ )
        ⟨ Process and Match Up Mesh Edge 21.9 ⟩
    else
        assert(( $num\_edges \equiv 0$ )  $\wedge$  ( $poly \equiv int\_undef$ ))
        /* Do nothing; didn’t invoke either dg_polygon or iedge. */
    end if
else
    if(( $process\_polygon \neq clear\_polygon$ )  $\wedge$  ( $n \equiv 0$ )  $\wedge$  ( $num\_edges \equiv 0$ )) then
        assert('Null $\sqcup$ polygon!'  $\equiv$  ' $\sqcup$ ')
    end if
end if

```

This code is used in section 21.

Look for a closed curve along the specified mesh edge(s) and turn it into a polygon.

```

⟨ Process Closed iedge Polygon 21.7 ⟩ ≡
  assert(poly ≡ int_undef)
    /* This is the only keyword specified for this polygon. We need to look for a closed curve. First,
       assemble the elements from the specified edge(s) into a single array. Note that the dimensions of
       mesh_edge_elements are ordered such that it cannot be used directly in an argument list. */
  i_tot = 0
  do i = 1, num_iedges
    do j = 1, mesh_edge_num_elements_iedges_i
      i_tot++
      mesh_scratch_i_tot = mesh_edge_elements_j,iedges_i
    end do
  end do  /* Determine how many open and closed curves are described by this set of elements. */
  call count_curves(mesh_elements, i_tot, mesh_scratch, num_curves, curve_type, curve_start,
                    curve_start_tip, mesh_curve_num)
  assert(num_curves ≥ 1)
  /* Go through the resulting list to see if any of them are closed. We need one and only one. If we
     encounter situations in which more than one is legitimately possible, need to allow for it. */
  closed_curve = 0
  do i = 1, num_curves
    if (curve_type_i ≡ curve_closed) then
      assert(closed_curve ≡ 0)
      closed_curve = i
    end if
  end do
  assert(closed_curve ≠ 0)
  /* Finally, convert the list of elements corresponding to that curve into a polygon. */
  call elements_to_polygon(mesh_nodes, mesh_elements, i_tot, mesh_scratch, curve_startclosed_curve,
                          curve_start_tipclosed_curve, n, polygon0,g2_x)
  /* Check to see if the polygon is clockwise (positive area); reverse it if not. */
  poly_area = polygon_area(n, polygon0,g2_x)
  if (poly_area < zero)
    call reverse_polygon(n, polygon0,g2_x)

```

This code is used in section 21.6.

Use the mesh connection information to identify a single point and add it to the existing polygon.

$\langle$  Process Single Mesh Point 21.8  $\rangle \equiv$

```

call get_mc_edge_seg(dg_poly_meshconpoly,1,1, dg_poly_meshcon_hvpoly,1,1, mesh_edge_num_elements,
    mesh_edge_dg_label, mesh_edge_hv, mc_edge1,1, mc_seg1,1) /* Collect the node numbers */
mc_node1 = mesh_elementsmesh_edge_elements mc_seg1,1,mc_edge1,1,element_start
mc_node2 = mesh_elementsmesh_edge_elements mc_seg1,1,mc_edge1,1,element_end
mc_node3 = mesh_elementsmesh_edge_elements mc_seg1,2,mc_edge1,2,element_start
mc_node4 = mesh_elementsmesh_edge_elements mc_seg1,2,mc_edge1,2,element_end
/* Find the common one */
if((mc_node1 ≡ mc_node3) ∨ (mc_node2 ≡ mc_node3)) then
    common_node = mc_node3
else if((mc_node1 ≡ mc_node4) ∨ (mc_node2 ≡ mc_node4)) then
    common_node = mc_node4
else
    assert('Mesh_connection_elements_do_not_have_a_common_node!' ≡ ' ')
end if /* Add to the polygon */
polygonn,g2_x = mesh_nodescommon_node,g2_x
polygonn,g2_z = mesh_nodescommon_node,g2_z
n++ /* It should now be "closed". The orientation is still uncertain, so we need to check it and
reverse if it's counter-clockwise. But, we need to explicitly fill in the last point to use these
routines since the above operations would not have set it. */
polygonn,g2_x = polygon0,g2_x
polygonn,g2_z = polygon0,g2_z
poly_area = polygon_area(n, polygon0,g2_x)
if(poly_area < zero)
    call reverse_polygon(n, polygon0,g2_x)

```

This code is used in section 21.6.

Use the *iedge* and mesh connection information to identify a mesh edge and connect it to the existing polygon.

```

⟨ Process and Match Up Mesh Edge 21.9 ⟩ ≡
  do  $i_{mc} = 1, 2$ 
    call  $get\_mc\_edge\_seg(dg\_poly\_meshcon_{poly,i_{mc},1}, dg\_poly\_meshcon_{hv,poly,i_{mc},1},$ 
          $mesh\_edge\_num\_elements, mesh\_edge\_dg\_label, mesh\_edge\_hv, mc\_edge_{i_{mc},1}, mc\_seg_{i_{mc},1})$ 
    /* Check order of the resulting edges and segments and swap if necessary so they conform to the
       convention for traversing the mesh edges. Also, verify that they are consecutive. */
    if ( $mod(mc\_edge_{i_{mc},2}, mesh\_num\_edges) \equiv mod((mc\_edge_{i_{mc},1} + 1), mesh\_num\_edges)$ ) then
      swap_edges = FALSE
      assert( $mc\_seg_{i_{mc},1} \equiv mesh\_edge\_num\_elements_{mc\_edge_{i_{mc},1}}$ )
      assert( $mc\_seg_{i_{mc},2} \equiv 1$ )
    else if ( $mod(mc\_edge_{i_{mc},2}, mesh\_num\_edges) \equiv mod((mc\_edge_{i_{mc},1} - 1), mesh\_num\_edges)$ ) then
      swap_edges = TRUE
      assert( $mc\_seg_{i_{mc},1} \equiv 1$ )
      assert( $mc\_seg_{i_{mc},2} \equiv mesh\_edge\_num\_elements_{mc\_edge_{i_{mc},2}}$ )
    else
      assert( $mc\_edge_{i_{mc},2} \equiv mc\_edge_{i_{mc},1}$ )
      if ( $mc\_seg_{i_{mc},2} \equiv mc\_seg_{i_{mc},1} + 1$ ) then
        swap_edges = FALSE
      else if ( $mc\_seg_{i_{mc},2} \equiv mc\_seg_{i_{mc},1} - 1$ ) then
        swap_edges = TRUE
      else
        assert('Bad_mesh_connection_data' ≡ ' ')
      end if
    end if
    if ( $swap\_edges \equiv TRUE$ ) then
      tmp_edge =  $mc\_edge_{i_{mc},1}$ 
      tmp_seg =  $mc\_seg_{i_{mc},1}$ 
       $mc\_edge_{i_{mc},1} = mc\_edge_{i_{mc},2}$ 
       $mc\_seg_{i_{mc},1} = mc\_seg_{i_{mc},2}$ 
       $mc\_edge_{i_{mc},2} = tmp\_edge$ 
       $mc\_seg_{i_{mc},2} = tmp\_seg$ 
    end if
  end do  /* The result of this is two possible paths around the mesh edge(s): 1,2 → 2,1 or 2,2 → 1,1.
             We use the iedge specification to choose between them. */
  if ( $((mc\_edge_{1,2} \equiv iedges_1) \wedge (mc\_edge_{2,1} \equiv iedges_{num\_iedges}))$ ) then
    j_init =  $mc\_seg_{1,2}$ 
    j_fin =  $mc\_seg_{2,1}$ 
  else if ( $((mc\_edge_{2,2} \equiv iedges_1) \wedge (mc\_edge_{1,1} \equiv iedges_{num\_iedges}))$ ) then
    j_init =  $mc\_seg_{2,2}$ 
    j_fin =  $mc\_seg_{1,1}$ 
  else
    assert('Mesh_connection_data_and_iedge_specification_inconsistent!' ≡ ' ')
  end if
  i_tot = 0
  do  $i = 1, num\_iedges$ 
    if ( $i \equiv 1$ ) then
      j_min = j_init
    else
      j_min = 1
    end if
    ...
  end do

```

```

end if
if ( $i \equiv num\_iedges$ ) then
     $j\_max = j\_fin$ 
else
     $j\_max = mesh\_edge\_num\_elements_{iedges_i}$ 
end if
do  $j = j\_min, j\_max$ 
     $i\_tot++$ 
     $mesh\_scratch_{i\_tot} = mesh\_edge\_elements_{j, iedges_i}$ 
end do
end do /* Convert these into a polygon. Have to use a temporary array since we may still need to
           invert the order to match up with the existing polygon. */
call elements_to_polygon( $mesh\_nodes$ ,  $mesh\_elements$ ,  $i\_tot$ ,  $mesh\_scratch$ , 1, element_start,  $n\_iedge$ ,
    iedge_temp $_{0,g2\_x}$ ) /* Now match up with the end points of the existing polygon. The default
        ordering has the last point of polygon getting connected to the first point of iedge_temp (segment
        "1" below) and, thus, the last point of iedge_temp connected to the first point of polygon (segment
        "3"). The other possible ordering is represented by the other two segments and corresponds to
        reversing iedge_temp. Use the shortest of the four segments to decide between the two orderings.
    */
 $vc\_set(poly\_p_1, polygon_{0,g2\_x}, zero, polygon_{0,g2\_z})$ 
 $vc\_set(poly\_p_2, polygon_{n-1,g2\_x}, zero, polygon_{n-1,g2\_z})$ 
 $vc\_set(iedge\_p_1, iedge\_temp_{0,g2\_x}, zero, iedge\_temp_{0,g2\_z})$ 
 $vc\_set(iedge\_p_2, iedge\_temp_{n\_iedge-1,g2\_x}, zero, iedge\_temp_{n\_iedge-1,g2\_z})$ 
 $vc\_difference(poly\_p_2, iedge\_p_1, delta)$ 
 $dist_1 = vc\_abs(delta)$ 
 $vc\_difference(poly\_p_1, iedge\_p_1, delta)$ 
 $dist_2 = vc\_abs(delta)$ 
 $vc\_difference(poly\_p_1, iedge\_p_2, delta)$ 
 $dist_3 = vc\_abs(delta)$ 
 $vc\_difference(poly\_p_2, iedge\_p_2, delta)$ 
 $dist_4 = vc\_abs(delta)$ 
 $min\_dist = \min(dist_1, dist_2, dist_3, dist_4)$ 

if (( $min\_dist \equiv dist_2$ )  $\vee$  ( $min\_dist \equiv dist_4$ )) then
    call reverse_polygon( $n\_iedge - 1$ , iedge_temp $_{0,g2\_x}$ )
end if /* Now add the mesh points to the polygon. */
do  $i = 0, n\_iedge - 1$ 
     $i\_tot = n + i$ 
    assert( $i\_tot \leq g2\_num\_points - 1$ )
     $polygon_{i\_tot,g2\_x} = iedge\_temp_{i,g2\_x}$ 
     $polygon_{i\_tot,g2\_z} = iedge\_temp_{i,g2\_z}$ 
end do
 $n = i\_tot + 1$  /* It should now be "closed". The orientation is still uncertain, so we need to check it
    and reverse if it's counter-clockwise. But, we need to explicitly fill in the last point to use these
    routines since the above operations would not have set it. */
 $polygon_{n,g2\_x} = polygon_{0,g2\_x}$ 
 $polygon_{n,g2\_z} = polygon_{0,g2\_z}$ 
 $poly\_area = polygon\_area(n, polygon_{0,g2\_x})$ 
if ( $poly\_area < zero$ )
    call reverse_polygon( $n, polygon_{0,g2\_x}$ )

```

## 22 Read data specifying a diagnostic

```
"definegeometry2d.f" 22 ≡
@m diag_loop #:0

⟨Functions and subroutines 8⟩ +≡
subroutine specify_diagnostic(nunit, stratum, solid, variable, tab_index, var_min, var_max, mult,
spacing)

implicit none_f77
implicit none_f90

integer nunit // Input
integer stratum, solid, variable, tab_index, spacing // Output
real var_min, var_max, mult

integer length, p, b, e // Local
character*LINELEN line, keyword

st_decls /* Defaults */
stratum = int_uninit
solid = TRUE
variable = sc_diag_unknown
tab_index = 0
var_min = zero
var_max = zero
mult = zero
spacing = sc_diag_spacing_unknown

diag_loop: continue
assert(read_string(nunit, line, length))
assert(length ≤ len(line))
length = parse_string(line(:length))
p = 0
assert(next_token(line, b, e, p))
keyword = line(b : e)

if (keyword ≡ 'stratum') then
  assert(next_token(line, b, e, p))
  stratum = read_integer(line(b : e))
  assert(stratum ≠ int_uninit)
  if (next_token(line, b, e, p)) then
    keyword = line(b : e)
    if (keyword ≡ 'nonsolid') then
      solid = FALSE
    else if (keyword ≡ 'solid') then
      solid = TRUE
    else
      write(stderr, *) 'Unexpected sector type', line(b : e)
      assert(F)
    end if
  end if

else if (keyword ≡ 'variable') then
```

```

⟨ Variable Keyword 22.1 ⟩

else if (keyword ≡ 'number') then
  assert(next_token(line, b, e, p))
  tab_index = read_integer(line(b : e))
  assert(tab_index > 0)

else if (keyword ≡ 'minimum') then
  assert(next_token(line, b, e, p))
  var_min = read_real(line(b : e))

else if (keyword ≡ 'maximum') then
  assert(next_token(line, b, e, p))
  var_max = read_real(line(b : e))

else if (keyword ≡ 'multiplier') then
  assert(next_token(line, b, e, p))
  mult = read_real(line(b : e))
  assert(mult ≠ zero)

else if (keyword ≡ 'spacing') then
  assert(next_token(line, b, e, p))
  if (line(b : e) ≡ 'linear') then
    spacing = sc_diag_spacing_linear
  else if (line(b : e) ≡ 'log') then
    spacing = sc_diag_spacing_log
  else
    write(stderr, *) 'Unexpected spacing', line(b : e)
    assert(F)
  end if

else if (keyword ≡ 'end_diagnostic') then
  assert(stratum ≠ int_uninit)
  if (variable ≠ sc_diag_unknown) then
    assert(tab_index > 0)
    assert(mult ≠ zero)
    assert(var_min ≠ var_max)
    assert(spacing ≠ sc_diag_unknown)
    if (spacing ≡ sc_diag_spacing_log) then
      assert(var_min * mult > zero)
      assert(var_max * mult > zero)
    end if
  end if
  return

end if

go to diag_loop

end

```

Specify the diagnostic variable.

```

⟨ Variable Keyword 22.1 ⟩ ≡
  assert(next_token(line, b, e, p))
  if (line(b : e) ≡ 'energy') then
    variable = sc_diag_energy
  if (next_token(line, b, e, p)) then
    if (line(b : e) ≡ 'J') then
      mult = one
    else if (line(b : e) ≡ 'eV') then
      mult = electron_charge
    else
      write(stderr, *) 'Unexpected energy unit', line(b : e)
      assert(F)
    end if
  else
    mult = electron_charge // Default to eV
  end if
  else if (line(b : e) ≡ 'angle') then
    variable = sc_diag_angle
  if (next_token(line, b, e, p)) then
    if (line(b : e) ≡ 'radians') then
      mult = one
    else if (line(b : e) ≡ 'degrees') then
      mult = PI / const(1.8, 2)
    else
      write(stderr, *) 'Unexpected angle unit', line(b : e)
      assert(F)
    end if
  else
    mult = PI / const(1.8, 2) // Default to degrees
  end if
  else
    write(stderr, *) 'Unexpected diagnostic variable', line(b : e)
    assert(F)
  end if

```

This code is used in section 22.

## 23 Fill in data for this zone

NOTE: the center point is set using the argument *center* from the first call to this routine for the current zone (i.e., the routine should be called for each polygon comprising a complex zone).

⟨ Functions and subroutines 8 ⟩ +≡

```

subroutine update_zone_info(zonearray, ind_x, ind_z, zone_type_array, n, polygon, center, y_div,
                               y_values)

implicit none_f77
zn_common // Common
gi_common
implicit none_f90

integer ind_x, ind_z, n, y_div // Input
real polygon0:g2_num_points-1,g2_x:g2_z, y_values0:y_div
integer zonearray0:y_div-1
character*(*) zone_type_array0:y_div-1

vc_decl(center)
integer iy // Local
real phi_mid, y_frac

vc_decls
gi_ext

do iy = 0, y_div - 1
  if (zn_volume(zonearrayiy) ≡ real_undef) then
    zn_volume(zonearrayiy) = zero
    zn_index(zonearrayiy, zi_ix) = ind_x
    zn_index(zonearrayiy, zi_iz) = ind_z
    zn_index(zonearrayiy, zi_iy) = iy
    zn_index(zonearrayiy, zi_ptr) = zonearray0
    if (zone_type_arrayiy ≡ "solid") then
      zn_type_set(zonearrayiy, zn_solid)
    else if (zone_type_arrayiy ≡ "exit") then
      zn_type_set(zonearrayiy, zn_exit)
    else if (zone_type_arrayiy ≡ "vacuum") then
      zn_type_set(zonearrayiy, zn_vacuum)
    else if (zone_type_arrayiy ≡ "plasma") then
      zn_type_set(zonearrayiy, zn_plasma)
    else if (zone_type_arrayiy ≡ "exit") then
      zn_type_set(zonearrayiy, zn_exit)
    else
      write (stderr, *) 'Unknown zone type:', zone_type_arrayiy
      assert( $\mathcal{F}$ )
    end if
    call set_zn_min_max(n, polygon0,g2_x, zonearrayiy, T) /* This isn't the same test that's
       used in process_polygon_plane to discern a 3-D case from a 2-D. Is it adequate? Explicitly
       reset only for non-symmetric cases to be consistent with previous usage. */
  if ((geometry_symmetry ≡ geometry_symmetry_plane_hw) ∨ (geometry_symmetry ≡
        geometry_symmetry_cyl_hw) ∨ (geometry_symmetry ≡ geometry_symmetry_cyl_section))
    then
      if (y_valuesiy < y_valuesiy+1) then

```

```

zone_min_zonearrayiy,2 = y_valuesiy
zone_max_zonearrayiy,2 = y_valuesiy+1
else if (y_valuesiy > y_valuesiy+1) then
    zone_max_zonearrayiy,2 = y_valuesiy
    zone_min_zonearrayiy,2 = y_valuesiy+1
end if
end if /* Think this will always produce something in the zone. Leave y value equal to zero
   in the symmetry cases so as to be consistent with previous usage. */
vc_copy(center, zone_center_zonearrayiy)
if (geometry_symmetry ≡ geometry_symmetry_plane_hw) then
    zone_center_zonearrayiy,2 = half * (y_valuesiy + y_valuesiy+1)
else if (geometry_symmetry ≡ geometry_symmetry_cyl_hw ∨ geometry_symmetry ≡
           geometry_symmetry_cyl_section) then
    assert(center2 ≡ zero)
    phi_mid = half * (y_valuesiy + y_valuesiy+1)
    zone_center_zonearrayiy,1 = center1 * cos(phi_mid)
    zone_center_zonearrayiy,2 = center1 * sin(phi_mid)
else
    assert((geometry_symmetry ≡ geometry_symmetry_oned) ∨ (geometry_symmetry ≡
           geometry_symmetry_plane) ∨ (geometry_symmetry ≡ geometry_symmetry_cylindrical))
end if
else
    /* This routine doesn't touch the y value on subsequent calls so that the above generalization
       to 3-D is sufficient. */
    call set_zn_min_max(n, polygon0,g2_x, zonearrayiy, F)
end if
if (geometry_symmetry ≡ geometry_symmetry_plane_hw ∨ geometry_symmetry ≡
      geometry_symmetry_cyl_hw ∨ geometry_symmetry ≡ geometry_symmetry_cyl_section) then
    y_frac = abs(y_valuesiy+1 - y_valuesiy) / (universal_cell_max2 - universal_cell_min2)
else
    y_frac = one
end if
zn_volume(zonearrayiy) = zn_volume(zonearrayiy) + y_frac * polygon_volume(n, polygon0,g2_x)
end do

return
end

```

## 24 Define universal cell

Copied the version in *composite.web* and extended to handle nearly symmetric 3-D cases.

*<Functions and subroutines 8> +≡*

```

subroutine universal_cell_3d(symmetry, min_corner, max_corner, vol) implicit none f77
gi_common
implicit none f90
integer symmetry // Input
real min_corner3, max_corner3
real vol // Output
integer face, face1, face2, surf_type, i // Local
real cos_y, sin_y
real x3, 0:4, apex3, a3, b3, x03, x13, x23, x33, coeff NCOEFFS, tx3, 4
gi_ext
external vector_compare
integer vector_compare

geometry_symmetry = symmetry

$DO (I, 1, 3)
{
assert(min_cornerI < max_cornerI);
}
vc_copy(min_corner, universal_cell_min)
vc_copy(max_corner, universal_cell_max)

i = 0
x1, i = min_corner1;
x2, i = zero;
x3, i = min_corner3;
i++
x1, i = min_corner1;
x2, i = zero;
x3, i = max_corner3;
i++
x1, i = max_corner1;
x2, i = zero;
x3, i = max_corner3;
i++
x1, i = max_corner1;
x2, i = zero;
x3, i = min_corner3;
i++
vc_copy(x0, xi)

if ((symmetry ≡ geometry_symmetry_cylindrical) ∨ (symmetry ≡
geometry_symmetry_cyl_hw) ∨ (symmetry ≡ geometry_symmetry_cyl_section))
then

do i = 0, 3
call conea(xi,1, xi+1,1, coeff, surf_type, apex)
face = define_surface(coeff, T)
if (face > 0) then

```

```

    vc_copy( $x_i$ ,  $surface\_points_{0, face}$ )
    vc_copy( $x_{i+1}$ ,  $surface\_points_{1, face}$ )
  end if
  call add_surface( $face$ , 0,  $\mathcal{T}$ )
end do

if ( $symmetry \equiv geometry\_symmetry\_cyl\_section$ ) then
   $cos\_y = \cos(min\_corner_2)$ 
   $sin\_y = \sin(min\_corner_2)$ 
  vc_set( $a, -sin\_y, cos\_y, zero$ )
  vc_set( $b, max\_corner_1 * cos\_y, max\_corner_1 * sin\_y, max\_corner_3$ )
  call plane( $b, a, coeff$ )
   $face1 = define\_surface(coeff, \mathcal{T})$ 
  call add_surface( $face1$ , 0,  $\mathcal{T}$ )
  /* Check to see that the two ends are genuinely distinct surfaces. */
  if ( $abs(max\_corner_2 - min\_corner_2 - PI) > epsilon\_angle$ ) then
     $cos\_y = \cos(max\_corner_2)$ 
     $sin\_y = \sin(max\_corner_2)$ 
    vc_set( $a, -sin\_y, cos\_y, zero$ )
    vc_set( $b, max\_corner_1 * cos\_y, max\_corner_1 * sin\_y, max\_corner_3$ )
    call plane( $b, a, coeff$ )
     $face2 = define\_surface(coeff, \mathcal{T})$ 
    call add_surface( $-face2$ , 0,  $\mathcal{T}$ )
  end if
end if

 $vol = half * (max\_corner_2 - min\_corner_2) * (max\_corner_1^2 - min\_corner_1^2) * (max\_corner_3 - min\_corner_3)$ 

else if (( $symmetry \equiv geometry\_symmetry\_plane$ )  $\vee$  ( $symmetry \equiv geometry\_symmetry\_oned$ )  $\vee$  ( $symmetry \equiv geometry\_symmetry\_plane\_hw$ )) then
  vc_set( $a, zero, one, zero$ )
  call plane( $min\_corner, a, coeff$ )
   $face1 = define\_surface(coeff, \mathcal{T})$ 
  call add_surface( $face1$ , 0,  $\mathcal{T}$ )
  call plane( $max\_corner, a, coeff$ )
   $face2 = define\_surface(coeff, \mathcal{T})$ 
  call add_surface( $-face2$ , 0,  $\mathcal{T}$ )
  if ( $symmetry \neq geometry\_symmetry\_plane\_hw$ ) then
    call init_identity( $tx$ )
    vc_set( $x0, zero, -(max\_corner_2 - min\_corner_2), zero$ )
    call geom_translate( $x0, tx$ )
    call add_transform( $face1, face2, tx$ )
    call invert( $tx, tx$ )
    call add_transform( $-face2, -face1, tx$ )
  end if

  do  $i = 0, 3$ 
    vc_copy( $x_i, x1$ )
    vc_copy( $x_{i+1}, x2$ )
    if ( $vector\_compare(x1, x2) > 0$ ) then
      vc_set( $x3, x1_1, one, x1_3$ )
    else
      vc_set( $x3, x2_1, one, x2_3$ )
    end if
    call planea( $x1, x3, x2, coeff$ )
  end do
end if

```

```
face = define_surface_a(coeff, x1, x2)
if (i ≡ 3)
    face1 = face
if (i ≡ 1)
    face2 = -face
if (face > 0) then
    vc_copy(x1, surface_points0, face)
    vc_copy(x2, surface_points1, face)
end if
call add_surface(face, 0, T)
end do

if (symmetry ≡ geometry_symmetry_oned) then
    call init_identity(tx)
    vc_set(x0, zero, zero, -(max_corner3 - min_corner3))
    call geom_translate(x0, tx)
    call add_transform(face1, face2, tx)
    call invert(tx, tx)
    call add_transform(-face2, -face1, tx)
end if

vol = (max_corner1 - min_corner1) * (max_corner2 - min_corner2) * (max_corner3 - min_corner3)
end if

universal_cell_vol = vol

return end
```

## 25 Find the centroid of a triangle

This is computed using the fact that the centroid (equivalently, the center of gravity and the intersection of the medians) is at trilinear coordinates  $\frac{1}{a} : \frac{1}{b} : \frac{1}{c}$ , where  $a$ ,  $b$ , and  $c$  are the lengths of the sides of the triangles (for additional information see <http://mathworld.wolfram.com/TrilinearCoordinates.html>).

```

⟨ Functions and subroutines 8 ⟩ +≡
subroutine triangle_centroid(triangle, center)
  implicit none_f77
  implicit none_f90

  real triangle0:3,g2_x:g2_z // Input
  vc_decl(center) // Output

  real area, k, a, b, c, alpha, gamma, den, lc // Local

  vc_decl(a_vertex)
  vc_decl(b_vertex)
  vc_decl(c_vertex)
  vc_decl(a_vec)
  vc_decl(b_vec)
  vc_decl(c_vec)
  vc_decl(a_hat)
  vc_decl(c_hat)
  vc_decl(bc_cross)
  vc_decl(c_perp)
  vc_decl(neg_y)
  vc_decl(m_test1)
  vc_decl(m_test2)

  vc_decls
    /* Associate point 0 with vertex A, 1 with B, and 2 with C. Here are the lengths of the sides. */
    vc_set(a_vertex, triangle0,g2_x, zero, triangle0,g2_z)
    vc_set(b_vertex, triangle1,g2_x, zero, triangle1,g2_z)
    vc_set(c_vertex, triangle2,g2_x, zero, triangle2,g2_z)
    /* Vectors for two of the sides and the corresponding unit vectors. */
    vc_difference(b_vertex, a_vertex, c_vec)
    vc_unit(c_vec, c_hat)
    vc_difference(c_vertex, b_vertex, a_vec)
    vc_unit(a_vec, a_hat)
    /* Perpendicular to c, define using cross product to be sure it's pointing the right direction. */
    vc_set(neg_y, zero, -one, zero)
    vc_cross(c_hat, neg_y, c_perp) /* Lengths of all 3 sides. */
    vc_difference(a_vertex, c_vertex, b_vec)
    a = vc_abs(a_vec)
    b = vc_abs(b_vec)
    c = vc_abs(c_vec) /* Area by using the cross product: */
    vc_cross(b_vec, c_vec, bc_cross)
    area = half * vc_abs(bc_cross) /* Use  $\alpha$ ,  $\beta$ ,  $\gamma$  to denote the trilinear coordinate values. A constant
       of proportionality k relates these to the distances of the points to the triangle sides. k can be
       determined for any particular set of coordinates via the area of the triangle.
```

$$k \equiv \frac{2\Delta}{a\alpha + b\beta + c\gamma},$$

In the case of the centroid with  $\alpha = 1/a$ , etc., the denominator is just 3. \*/  
 $\alpha = \text{one} / a$   
 $\gamma = \text{one} / c$   
 $k = \text{two} * \text{area} / \text{const}(3)$  /\* The vectors used in the Web page referred to in the introduction to this routine do not appear to be a consistent set. The vectors we use here along the sides trace out the triangle in a clockwise direction. The Cartesian coordinates of the point of interest are written in terms of some distance  $l_c$  along side  $c$  (starting at vertex A) plus a distance  $k\gamma$  along  $\hat{c}_\perp$  (by definition of  $\gamma$ ). A similar expression is written going from vertex C along  $-\hat{a}$ , and the two are solved for the two distances along the sides. We only need one here:

$$l_c = \frac{-k\alpha + \gamma k(\hat{a}_1\hat{c}_1 + \hat{a}_3\hat{c}_3) + \hat{a}_3(A_1 - C_1) + \hat{a}_1(C_3 - A_3)}{\hat{a}_1\hat{c}_3 - \hat{a}_3\hat{c}_1}.$$

**Note:**  $l_c$  can be negative for a really stretched out triangle. \*/  
 $\text{den} = \text{a\_hat}_1 * \text{c\_hat}_3 - \text{a\_hat}_3 * \text{c\_hat}_1$   
 $\text{assert}(\text{den} > \text{zero})$   
 $\text{lc} = (-k * \alpha + \gamma * k * (\text{a\_hat}_1 * \text{c\_hat}_1 + \text{a\_hat}_3 * \text{c\_hat}_3) + \text{a\_hat}_3 * (\text{a\_vertex}_1 - \text{c\_vertex}_1) + \text{a\_hat}_1 * (\text{c\_vertex}_3 - \text{a\_vertex}_3)) / \text{den}$   
/\* Finally, the Cartesian coordinates are given in general by

$$\vec{x} = \vec{A} + l_c \hat{c} + k\gamma \hat{c}_\perp.$$

\*/  
 $\text{vc\_xvt}(\text{a\_vertex}, \text{c\_hat}, \text{lc}, \text{center})$   
 $\text{vc\_xvt}(\text{center}, \text{c\_perp}, k * \gamma, \text{center})$   
/\* Verify that this point is indeed inside the triangle. \*/  
 $\text{vc\_difference}(\text{center}, \text{a\_vertex}, \text{m\_test1})$   
 $\text{vc\_cross}(\text{m\_test1}, \text{c\_vec}, \text{m\_test2})$   
 $\text{assert}(\text{vc\_product}(\text{m\_test2}, \text{neg\_y}) > \text{zero})$   
  
 $\text{vc\_difference}(\text{center}, \text{b\_vertex}, \text{m\_test1})$   
 $\text{vc\_cross}(\text{m\_test1}, \text{a\_vec}, \text{m\_test2})$   
 $\text{assert}(\text{vc\_product}(\text{m\_test2}, \text{neg\_y}) > \text{zero})$   
  
 $\text{vc\_difference}(\text{center}, \text{c\_vertex}, \text{m\_test1})$   
 $\text{vc\_cross}(\text{m\_test1}, \text{b\_vec}, \text{m\_test2})$   
 $\text{assert}(\text{vc\_product}(\text{m\_test2}, \text{neg\_y}) > \text{zero})$   
  
**return**  
**end**

## 26 Compute the center of a quadrilateral

```

⟨ Functions and subroutines 8 ⟩ +≡
subroutine quad_center(quad, center)

implicit none_f77
implicit none_f90

real quad0:4,g2_x:g2_z // Input

vc_decl(center)
// Output /* Use this as a temporary approximation. To be useful, zone must be convex.
vc_set(center, const(0.25) * (quad0,g2_x + quad1,g2_x + quad2,g2_x + quad3,g2_x), zero,
      const(0.25) * (quad0,g2_z + quad1,g2_z + quad2,g2_z + quad3,g2_z))

return
end

```

## 27 Compute area of a closed polygon

This routine computes the area of a plane polygon using the formula employed in *polygon.volume*. That routine deals with the more general problem of computing the volume in both rectilinear and cylindrical geometries. The intended application of this more basic routine is to determine whether the points of a polygon are ordered clockwise (positive result) or counterclockwise (negative). As such, a simpler, self-contained routine is suitable.

There are  $n + 1$  points in the polygon. The numbering starts at 0 and ends at  $n$ . The coincidence of those two points is checked. This formula requires few other assumptions. Duplicate points can appear. Segments may be colinear. “Holes” can even be accounted for as long as they are specified as part of a single list of connected points. I.e., this is the same manner used for specifying for Triangle polygons with holes.

```

⟨ Functions and subroutines 8 ⟩ +≡
function polygon_area(n, x)

implicit none_f77
implicit none_f90

real polygon_area // Function
integer n // Input
real x0:n,g2_x:g2_z
integer i // Local
real area /* Use a local variable to facilitate debugging. */

assert(x0,g2_x ≡ xn,g2_x)
assert(x0,g2_z ≡ xn,g2_z)
area = zero
do i = 0, n - 1
  area += half * (xi,g2_x + xi+1,g2_x) * (xi,g2_z - xi+1,g2_z)
end do

polygon_area = area

return
end

```

## 28 Reverse order of polygon points

The objective of this routine is to transform a counter-clockwise oriented polygon into one with clockwise orientation. While those designations make sense only for closed polygons, the routine will, of course, work on an arbitrary list of points. The input integer  $n$  gives the number of sides of the polygon; there are  $n + 1$  points, numbered 0 through  $n$ .

```
( Functions and subroutines 8 ) +≡
subroutine reverse_polygon( $n$ ,  $x$ )
  implicit none_f77
  implicit none_f90

  integer  $n$  // Input
  real  $x_{0:n,g2\_x:g2\_z}$ 
  integer  $i$  // Local
  real  $x\_temp_{g2\_x:g2\_z}$  /* This should work for both even and odd  $n$ . */

  do  $i = 0, (n - 1) / 2$ 
     $x\_temp_{g2\_x} = x_{n-i,g2\_x}$ 
     $x\_temp_{g2\_z} = x_{n-i,g2\_z}$ 
     $x_{n-i,g2\_x} = x_{i,g2\_x}$ 
     $x_{n-i,g2\_z} = x_{i,g2\_z}$ 
     $x_{i,g2\_x} = x\_temp_{g2\_x}$ 
     $x_{i,g2\_z} = x\_temp_{g2\_z}$ 
  end do

  return
end
```

## 29 Find endpoints in a set of elements

This routine examines a set of DG-type elements (*poly\_elements*, consisting of *n* elements), presumably connected, to find the nodes belonging to a single element, i.e., the endpoints of the element chain. The number of endpoints *num\_ends* is returned, as are the corresponding node numbers (*end\_nodes*), pointers into the input *poly\_elements* list (*end\_elements*), and tips of those elements (*end\_tips*).

```

< Functions and subroutines 8 > +≡
subroutine find_endpoints(elements_list, n, poly_elements, num_ends, end_nodes, end_elements,
                           end_tips)
implicit none_f77
implicit none_f90

integer n      // Input
integer elements_list(*,element_start:element_end, poly_elements_n

integer num_ends    // Output
integer end_nodes_*, end_elements_*, end_tips_*

integer i, tip, test_node, match, j    // Local

num_ends = 0
do i = 1, n
  do tip = element_start, element_end
    test_node = elements_list(poly_elements_i,tip
    match = FALSE
    do j = 1, n
      if ((j ≠ i) ∧ ((elements_list(poly_elements_j,element_start ≡
                                     test_node) ∨ (elements_list(poly_elements_j,element_end ≡ test_node))) then
        /* Should only be one match! */
        assert(match ≡ FALSE)
        match = TRUE
      end if
    end do
    if (match ≡ FALSE) then
      num_ends++
      end_nodes_num_ends = test_node
      end_elements_num_ends = i
      end_tips_num_ends = tip
    end if
  end do
end do /* Number of endpoints should be even! */
assert((2 * (num_ends / 2) - num_ends) ≡ 0)
return
end

```

## 30 Convert elements to a polygon

This routine transforms a set of DG-type elements, *poly\_elements*, into a linear list of points, a “polygon” as used elsewhere in this code, *poly\_x*, consisting of *poly\_num\_points*. The elements need not be all consistently oriented (i.e., so that each interior node is the “end” of one element and the “start” of another). But, the routine does expect to finish with a single chain (two endpoints) or a closed polygon. In the former case, the input pointer to the initial element (*initial\_element*) and tip (*initial\_tip*) must correspond to one of those endpoints.

```
"definegeometry2d.f" 30 ==
@m point_loop #:0

⟨ Functions and subroutines 8 ⟩ +≡
subroutine elements_to_polygon(nodes, elements_list, n, poly_elements, initial_element, initial_tip,
    poly_num_points, poly_x)
implicit none_f77
implicit none_f90

integer n, initial_element, initial_tip      // Input
integer elements_list*,element_start:element_end, poly_elements_n
real nodes*,g2_x:g2_z

integer poly_num_points      // Output
real poly_x*,g2_x:g2_z

integer i, j_match, node, tip, j, match, tip_match    // Local /* The task here is to determine the
                                                       desired ordering of the nodes in the input element set, i.e., their connectivity. We assume that
                                                       the caller has provided the initial point in the output polygon. We then try to find a node of
                                                       another element in the set that matches its opposite tip. If we do, we add it to the output
                                                       polygon. The point_loop continues in this way until we are left with a point that has no match.
                                                       This is then one end of the open loop. If that doesn't occur (j_match = 1), the loop is closed.

assert(initial_element ≥ 1)
assert(initial_element ≤ n)
assert((initial_tip ≡ element_start) ∨ (initial_tip ≡ element_end))
poly_num_points = 0
i = initial_element
j_match = 0
node = elements_list poly_elements initial_element,initial_tip
poly_x poly_num_points,g2_x = nodes node,g2_x
poly_x poly_num_points,g2_z = nodes node,g2_z
poly_num_points++ /* This restriction may not apply in general. Leave for now until we can
                   replace it with a corresponding assertion in the calling program. */
assert(poly_num_points ≤ g2_num_points - 1)
if (initial_tip ≡ element_start) then
    tip = element_end
else
    tip = element_start
end if

point_loop: continue
node = elements_list poly_elements i,tip
poly_x poly_num_points,g2_x = nodes node,g2_x
poly_x poly_num_points,g2_z = nodes node,g2_z
poly_num_points++ /* See above comment */
```

```

assert(poly_num_points ≤ g2_num_points - 1)
match = FALSE
/* Look through all of the other elements in the DG polygon for a matching point. Need to try
both ends of each candidate element! For this reason, must exclude current element. */
do j = 1, n
  if ((j ≠ i) ∧ (elements_listpoly_elementsj,element_start ≡ elements_listpoly_elementsi,tip)) then
    assert(match ≡ FALSE)
    match = TRUE
    j_match = j
    tip_match = element_end
  else if ((j ≠ i) ∧ (elements_listpoly_elementsj,element_end ≡ elements_listpoly_elementsi,tip)) then
    assert(match ≡ FALSE)
    match = TRUE
    j_match = j
    tip_match = element_start
  end if
end do
if (j_match ≠ initial_element) then
  if (match ≡ TRUE) then /* Got a match. Reset the bookkeeping indices so that the node gets
added to the output polygon and we can begin the search again for a match to the node at
the other end of this DG polygon element. */
    i = j_match
    j_match = 0
    tip = tip_match
    go to point_loop
  /* Otherwise, we have reached the other end point. Check that the number of points in the
polygon is as expected. As is the convention elsewhere in this code, the poly_num_points
entry in the poly_x array is reserved for the next entry in the array. The original usage of
this algorithm was for cases using all of the input points, corresponding to the equality. */
else
  assert((poly_num_points - 1) ≤ n)
end if /* If the match is with the initial_element, we are also done. In this case, though, we
need to reduce poly_num_points by 1 since this is already a closed polygon (first and last
points are same). */
else
  assert((poly_num_points - 1) ≤ n)
  poly_num_points --
end if
return
end

```

## 31 Count the number of continuous sections in set of elements

Namely, a set of elements should consist of one or more continuous sections, or curves. Some of these will be “open”, i.e., having two unmatched endpoints. The others will be “closed”. The primary objective of this routine is to find the latter. The routine returns the number of curves found and characterizes each as “open” or “closed”. For convenience, a second array identifies an element belonging to that curve, *curve\_start*. Another array labels each element according to the curve it belongs to.

*<Functions and subroutines 8> +≡*

```

subroutine count_curves(elements_list, n, poly_elements, num_curves, curve_type, curve_start,
    curve_start_tip, element_curve_num)
implicit none_f77
implicit none_f90

integer n // Input
integer elements_list*,element_start:element_end, poly_elements_n
integer num_curves // Output
integer curve_type dim_curves, curve_start dim_curves, curve_start_tip dim_curves,
    element_curve_num*
integer i, i_init, j_match, tip, node, match, j, tip_match, /* Local */
    num_ends, i_tot
    /* Dimensioning these as in specify_polygon. Yes, these are ridiculously large. */
integer end_nodes2*g2_num_points, end_elements2*g2_num_points, end_tips2*g2_num_points
num_curves = 0
assert(n > 1) /* Initially label each curve with “0” to indicate that it hasn’t been checked. */
do i = 1, n
    element_curve_numi = 0
end do /* By making a little extra effort, we can get all of the open curves correctly labeled, as
    well as finding the closed ones. The idea is to start with “endpoint” elements first, finding them
    with find_endpoints: */
call find_endpoints(elements_list, n, poly_elements, num_ends, end_nodes, end_elements, end_tips)
assert(num_ends ≤ 2 * g2_num_points)
    /* We will add them at the beginning of the main loop. In this case, we take care to get the tip
    right. The rest of the loop is over all remaining elements. Presumably, these are all on one or
    more closed curves. For these, the choice of tip should not matter. In either case, if an element
    has not already been labeled with a curve number, it must belong to a new curve! */
do i_tot = 1, n + num_ends
    if (i_tot ≤ num_ends) then
        i_init = end_elementsi_tot
        if (end_tipsi_tot ≡ element_start) then
            tip = element_end
        else
            tip = element_start
        end if
    else
        i_init = i_tot - num_ends
        tip = element_end // Arbitrarily start with this tip
    end if
    if (element_curve_numi_init ≡ 0) then
        num_curves ++
        assert(num_curves ≤ dim_curves) /* At this point, the logic closely follows the “forward
        loop” procedure used in elements_to_polygon. */
    
```

```

i = i_init
element_curve_numi = num_curves
    /* Note that curve_start need not be the first point on an open curve. The real need is
       for a point on a closed curve. Recall that the other end of the current element is already
       considered to be on the curve and, thus, represents the actual starting tip. */
curve_start_num_curves = i
if (tip ≡ element_start) then
    curve_start_tip_num_curves = element_end
else
    curve_start_tip_num_curves = element_start
end if
j_match = 0

point_loop: continue
node = elements_listpoly_elementsi,tip
match = FALSE /* Look through all of the other non-labeled elements in the DG polygon
   for a matching point. Need to try both ends of each candidate element! For this reason,
   must exclude current element. */
do j = 1, n
    if ((j ≠ i) ∧ (elements_listpoly_elementsj,element_start ≡ elements_listpoly_elementsi,tip)) then
        assert(match ≡ FALSE)
        match = TRUE
        j_match = j
        tip_match = element_end
    else if ((j ≠ i) ∧ (elements_listpoly_elementsj,element_end ≡ elements_listpoly_elementsi,tip))
        then
            assert(match ≡ FALSE)
            match = TRUE
            j_match = j
            tip_match = element_start
        end if
    end do
    if (match ≡ FALSE) then /* Reached the end of an open curve. Can go to next i_init. */
        curve_typenum_curves = curve_open
    else
        if (j_match ≠ i_init) then /* Got a match that's not the initial element. Label the new
           element, reset the bookkeeping, and go through the point_loop again. */
            assert(element_curve_numj_match ≡ 0)
            element_curve_numj_match = num_curves
            i = j_match
            j_match = 0
            tip = tip_match
            go to point_loop
        else /* Got a match and it is the initial element. Label the curve accordingly and go to
           next i_init. */
            curve_typenum_curves = curve_closed
        end if
    end if
else
    assert(element_curve_numi_init > 0)
    assert(element_curve_numi_init ≤ num_curves)
end if

```

```

end do
assert(num_curves > 0)
return
end

```

## 32 Search mesh edges to find the ones corresponding to the input DG label

```

⟨Functions and subroutines 8⟩ +≡
subroutine get_mc_edge_seg(meshcon, meshcon_hv, mesh_edge_num_elements, mesh_edge_dg_label,
    mesh_edge_hv, mc_edge, mc_seg)

implicit none_f77
implicit none_f90

integer meshcon1:meshcon_elem_max, /* Input */
    meshcon_hv1:meshcon_elem_max, mesh_edge_num_elementsmesh_num_edges,
    mesh_edge_dg_label*,mesh_num_edges, mesh_edge_hvmesh_num_edges

integer mc_edge1:meshcon_elem_max, mc_seg1:meshcon_elem_max // Output
integer i_elem, i_edge, j_seg // Local

do i_elem = 1, meshcon_elem_max
    mc_edgei_elem = int_undef
    mc_segi_elem = int_undef
    do i_edge = 1, mesh_num_edges
        if (mesh_edge_hvi_edge ≡ meshcon_hvi_elem) then
            do j_seg = 1, mesh_edge_num_elementsi_edge
                if (meshconi_elem ≡ mesh_edge_dg_labelj_seg,i_edge) then
                    mc_edgei_elem = i_edge
                    mc_segi_elem = j_seg
                end if
            end do
        end if
    end do
    assert(mc_edgei_elem ≠ int_undef)
    assert(mc_segi_elem ≠ int_undef)
end do

return
end

```

### 33 Attempt to find the closest point on the two input edges to the input point

The full set of mesh data are input as are two of its edges, *iedge1* and *iedge2*. The routine first finds the closest point on each edge to the input point, *poly-p*. It then compares the distance from each to *poly-p* and returns the closer one. Namely, *edge\_chosen* is set to *iedge1* or *iedge2* accordingly.

*<Functions and subroutines 8> +≡*

```

subroutine match_edge_to_poly(vc_dummy(poly_p), mesh_nodes, mesh_elements,
    mesh_edge_num_elements, mesh_edge_elements, iedge1, iedge2, mesh_scratch, edge_chosen,
    closest_node, closest_elements, closest_tips)

implicit none f77
implicit none f90

integer iedge1, iedge2 // Input
integer mesh_elements*,element_start:element_end, mesh_edge_num_elementsmesh_num_edges,
    mesh_edge_elements*,mesh_num_edges, mesh_scratch*
real mesh_nodes*,g2_x:g2_z
vc_decl(poly_p)

integer edge_chosen, closest_node // Output
integer closest_elements2, closest_tips2

integer i, j
integer iedges2, close_node2, close_elements2,2, close_tips2,2
real dist_p_edge2

vc_decl(edge_p2)
vc_decl(delta_p2)

iedges1 = iedge1
iedges2 = iedge2
do i = 1, 2
    do j = 1, mesh_edge_num_elementsiedgesi
        mesh_scratchj = mesh_edge_elementsj,iedgesi
    end do
    call closest_point(vc_args(poly_p), mesh_nodes, mesh_elements, mesh_edge_num_elementsiedgesi,
        mesh_scratch, close_nodei, close_elementsi,1, close_tipsi,1)
    vc_set(edge_pi, mesh_nodesclose_nodei,g2_x, zero, mesh_nodesclose_nodei,g2_z)
    vc_difference(edge_pi, poly_p, delta_pi)
    dist_p_edgei = vc_abs(delta_pi)
end do
if (dist_p_edge1 ≤ dist_p_edge2) then
    i = 1
else
    i = 2
end if

edge_chosen = iedgesi
closest_node = close_nodei
do j = 1, 2
    closest_elementsj = close_elementsi,j
    closest_tipsj = close_tipsi,j

```

§33 [#42] **definegeometry2d** Attempt to find the closest point on the two input edges to the input point 112

```
end do  
return  
end
```

### 34 Find closest point in a set of elements, *edge\_elements*, to the point provided, *poly\_p*

*closest\_node* is returned as a pointer to the node representing that point. This node may belong to one or two elements, *closest\_elements*, with the corresponding tips given in *closest\_tips*. If it's just one element, the two values of *closest\_elements* and *closest\_tips* are set equal to one another.

*(Functions and subroutines 8) +≡*

```

subroutine closest_point(vc_dummy(poly_p), mesh_nodes, mesh_elements, edge_num_elements,
                           edge_elements, closest_node, closest_elements, closest_tips)
implicit none f77
implicit none f90

integer edge_num_elements // Input
integer mesh_elements*,element_start:element_end, edge_elements*
real mesh_nodes*,g2_x:g2_z
vc_decl(poly_p)

integer closest_node // Output
integer closest_elements2, closest_tips2

integer num_closest, i, j
real closest_distance, dist

vc_decl(edge_point)
vc_decl(delta) /* Initialize parameters */
closest_distance = geom_infinity
num_closest = 0 /* Loop over elements and the two nodes of each */
do i = 1, edge_num_elements
    do j = element_start, element_end
        vc_set(edge_point, mesh_nodesmesh_elements edge_elements i,j,g2_x, zero,
                  mesh_nodesmesh_elements edge_elements i,j,g2_z)
        vc_difference(poly_p, edge_point, delta)
        dist = vc_abs(delta)
        /* If this point is closer to poly_p than any other before, reset all of the output variables. */
        if (dist < closest_distance) then
            closest_node = mesh_elements edge_elements i,j
            closest_elements1 = i
            closest_tips1 = j
            num_closest = 1
            closest_distance = dist /* If the distance is equal to the closest one so far, this element
                                         should be sharing the node with the one that gave rise to the current closest_distance.
                                         This is checked by the assertion. */
            assert(mesh_elements edge_elements i,j ≡ closest_node)
            closest_elements2 = i
            closest_tips2 = j
            num_closest = 2
        end if
    end do
end do
assert(num_closest > 0) /* If the closest point is an end point, it will only be on one element. In
                                this case, equate the two entries so we don't have to keep track of how many there are. */

```

```
if (num_closest == 1) then
    closest_elements_2 = closest_elements_1
    closest_tips_2 = closest_tips_1
end if
return
end
```

## 35 Set up sectors

This code is based on the original in *readgeometry.web*. An extension to handle user-specified, auxiliary sectors has been added.

```

⟨ Functions and subroutines 8 ⟩ +≡
subroutine setup_sectors(num_polygons, polygon_points, polygon_xz, polygon_segment,
polygon_zone, poly_int_props, poly_real_props, num_aux_sectors, aux_stratum,
aux_stratum_poly, aux_stratum_points, aux_stratum_segment, y_div, sect_zone1, sect_zone2)
implicit none_f77
zn_common // Common
sc_common
implicit none_f90

integer num_polygons, num_aux_sectors, y_div // Input
integer polygon_segment*,0:g2_num_points-1, polygon_points*, polygon_zone*,
poly_int_props*,1:poly_int_max, aux_stratum*, aux_stratum_poly*, aux_stratum_points*,
aux_stratum_segment*
real polygon_xz*,0:g2_num_points-1,g2_x:g2_z, poly_real_props*,1:poly_real_max
integer sect_zone1*, sect_zone2* // Scratch
integer i_poly, j, sector1, face1, i_aux, /* Local */
num_zone1, num_zone2, k_zone2, k_zone1, side1_done, aux1_done
vc_decl(x1)
vc_decl(x2)

⟨ Memory allocation interface 0 ⟩

gi_ext /* Set up default sectors. These are taken to be at the interface between plasma / vacuum
and solid / exit zones. The procedure consists of cycling through all of the segments of all of
the polygons. The find_poly_zone routine returns all zones on both sides of this segment. In
symmetric problems, there is just one on each side. In problems with resolution in the third
dimension, more than one can appear on each side. When there are multiple entries on each
side, the index for the third dimension in zn_index is used to identify zones in the same plane.
find_poly_zone arranges for the type of zone1 to match that of the polygon. Then, sectors are
defined at adjacent zones of different types. For example, a target sector is identified by having
zone1 be a solid and zone2 be plasma. The target sector is then associated with zone1 (and its
corresponding face, face1); zone1 is passed to define_sector as a check. */
do i_poly = 1, num_polygons
  do j = 0, polygon_pointsi_poly - 1
    if (polygon_xzi_poly,j,g2_x ≠ polygon_xzi_poly,j+1,g2_x ∨ polygon_xzi_poly,j,g2_z ≠
      polygon_xzi_poly,j+1,g2_z) then /* This used to be part of find_poly_zone. Note the
      explicit insertion of zero here to match the value used in the process_polygon routines. */
      vc_set(x1, polygon_xz_{i_poly,j,g2_x}, zero, polygon_xz_{i_poly,j,g2_z})
      vc_set(x2, polygon_xz_{i_poly,j+1,g2_x}, zero, polygon_xz_{i_poly,j+1,g2_z})
      face1 = lookup_surface(x1, x2) /* The find_poly_zone routine ensures that zones associated
      with the current polygon are in sect_zone1; the adjacent zones are in sect_zone2. The
      sign of face1 is such that it is a face of the sect_zone1 zones. That is, flights leaving one
      of these zones will be going out through face1. */
      call find_poly_zone(face1, zn_type(polygon_zone_{i_poly}), polygon_zone_{i_poly}, y_div,
        sect_zone1, num_zone1, sect_zone2, num_zone2)
      side1_done = FALSE
      aux1_done = FALSE

```

```

if ((num_zone1 > 0)  $\wedge$  (num_zone2 > 0)) then
  do k_zone1 = 1, num_zone1
    do k_zone2 = 1, num_zone2 /* The test used in define_sector is not going to be
      able to verify that these two zones are indeed adjacent, only that they share
      a (toroidally continuous) common surface. Instead, we have to rely on the
      bookkeeping we do here to ensure that they are in fact neighbors. */
  @#if 0
    if ((num_zone1  $\equiv$  1)  $\vee$  (zn_index(sect_zone2k_zone2,
      zi_iy)  $\equiv$  zn_index(sect_zone1k_zone1, zi_iy))) then
  @#else
    if ((num_zone1  $\neq$  num_zone2)  $\vee$  (zn_index(sect_zone2k_zone2,
      zi_iy)  $\equiv$  zn_index(sect_zone1k_zone1, zi_iy))) then
  @#endif
    /* TARGET */
    if ((zn_type(sect_zone1k_zone1)  $\equiv$  zn_solid)  $\wedge$  (zn_type(sect_zone2k_zone2)  $\equiv$ 
      zn_plasma)) then
      /* This is the same sector we defined in the previous 2-D only version. The
       main difference is that now we're calling it a plasma sector. Believe that
       this assertion should be true based on the way solid zones are defined. Use
       here and below just in case to prevent duplicate plasma sectors. */
      assert((num_zone1  $\equiv$  1)  $\vee$  (num_zone1  $\equiv$  num_zone2))
      sector1 = define_sector(poly_int_propsi_poly, poly_stratum,
        polygon_segmenti_poly,j, -face1, sect_zone2k_zone2, sect_zone1k_zone1)
      define_sector_plasma(sector1)
      /* This is then the sector based on the zone on the other side. I.e., this zone
       is part of polygon i_poly. In principle, we could have the if-then statement
       check for zone1 being type plasma and zone2 being type solid, but then we
       wouldn't have a temperature associated with that polygon. If num_zone1
       = 1 and num_zone2 > 1, we could have multiple sectors on the plasma
       side, but would need just one on the solid side. The flag side1_done is used
       to prevent duplicate sectors from being defined on the solid side. This
       basically applies to every surface and zone1 combination so that the flag
       needs to be reset with each call to find_poly_zone and needs to be checked
       in each of the sections (since the type of zone2 may vary with k_zone2). */
    if ((num_zone1 > 1)  $\vee$  (side1_done  $\equiv$  FALSE)) then
      sector1 = define_sector(poly_int_propsi_poly, poly_stratum,
        polygon_segmenti_poly,j, face1, sect_zone1k_zone1, sect_zone2k_zone2)
      define_sector_target(sector1, poly_int_propsi_poly, poly_material,
        poly_real_propsi_poly, poly_temperature * boltzmanns_const,
        poly_real_propsi_poly, poly_recyc_coef)
      side1_done = TRUE
    end if /* WALL */
  else if ((zn_type(sect_zone1k_zone1)  $\equiv$  zn_solid)  $\wedge$  (zn_type(sect_zone2k_zone2)  $\equiv$ 
    zn_vacuum)) then
    assert((num_zone1  $\equiv$  1)  $\vee$  (num_zone1  $\equiv$  num_zone2))
    sector1 = define_sector(poly_int_propsi_poly, poly_stratum,
      polygon_segmenti_poly,j, -face1, sect_zone2k_zone2, sect_zone1k_zone1)
      /* Do not currently have a "temperature" associated with vacuum sectors.
       May need one for gas puffing there. */
    define_sector_vacuum(sector1)
  if ((num_zone1 > 1)  $\vee$  (side1_done  $\equiv$  FALSE)) then

```

```

sector1 = define_sector(poly_int_propsi_poly,poly,stratum,
    polygon_segmenti_poly,j, face1, sect_zone1k_zone1, sect_zone2k_zone2)
define_sector_wall(sector1, poly_int_propsi_poly,poly,material,
    poly_real_propsi_poly,poly,temperature * boltzmanns_const,
    poly_real_propsi_poly,poly,recyc_coef)
side1_done = TRUE
end if /* EXIT - PLASMA */
else if ((zn_type(sect_zone1k_zone1) ≡ zn_exit) ∧ (zn_type(sect_zone2k_zone2) ≡
    zn_plasma)) then
    assert((num_zone1 ≡ 1) ∨ (num_zone1 ≡ num_zone2))
    sector1 = define_sector(poly_int_propsi_poly,poly,stratum,
        polygon_segmenti_poly,j, -face1, sect_zone2k_zone2, sect_zone1k_zone1)
define_sector_plasma(sector1)
if ((num_zone1 > 1) ∨ (side1_done ≡ FALSE)) then
    sector1 = define_sector(poly_int_propsi_poly,poly,stratum,
        polygon_segmenti_poly,j, face1, sect_zone1k_zone1, sect_zone2k_zone2)
define_sector_exit(sector1)
    side1_done = TRUE
end if /* EXIT - VACUUM */
else if ((zn_type(sect_zone1k_zone1) ≡ zn_exit) ∧ (zn_type(sect_zone2k_zone2) ≡
    zn_vacuum)) then
    assert((num_zone1 ≡ 1) ∨ (num_zone1 ≡ num_zone2))
    sector1 = define_sector(poly_int_propsi_poly,poly,stratum,
        polygon_segmenti_poly,j, -face1, sect_zone2k_zone2, sect_zone1k_zone1)
define_sector_vacuum(sector1)
if ((num_zone1 > 1) ∨ (side1_done ≡ FALSE)) then
    sector1 = define_sector(poly_int_propsi_poly,poly,stratum,
        polygon_segmenti_poly,j, face1, sect_zone1k_zone1, sect_zone2k_zone2)
define_sector_exit(sector1)
    side1_done = TRUE
end if
end if /* Define non-default sectors. The idea is to just define a sector
associated with a specific polygon segment. This sector is defined with the
same orientation as the wall and target sectors above. I.e., a flight leaving the
current polygon will be going out through this sector. Note that aux1_done
works just like side1_done does above. */
if (num_aux_sectors > 0) then
    do i_aux = 1, num_aux_sectors // AUX.
        if ((aux_stratum_polyi_aux ≡ i_poly) ∧ (aux_stratum_pointsi_aux ≡
            j) ∧ ((num_zone1 > 1) ∨ (aux1_done ≡ FALSE))) then
            sector1 = define_sector(aux_stratumi_aux, aux_stratum_segmenti_aux,
                face1, sect_zone1k_zone1, sect_zone2k_zone2)
            aux1_done = TRUE
        end if
    end do // num_aux_sectors
end if // zones line up
end if
end do // num_zone2
end do // num_zone1
end if // num_zone1, num_zone2 > 0
end if

```

```
    end do // polygon_points
end do // num_polygons
return
end
```

## 36 Print data from Triangle

This is taken directly from the *tricall* example program that is provided with Triangle.

```
< C Functions 36 >C ≡
void report(io, markers, reporttriangles, reportneighbors, reportsegments, reportedges,
            reportnorms, reportholes)
struct triangulateio *io;
int markers;
int reporttriangles;
int reportneighbors;
int reportsegments;
int reportedges;
int reportnorms;
int reportholes;
{
int i, j;
for (i = 0; i < io→numberofpoints; i++)
{
    printf("Point %4d:", i);
    for (j = 0; j < 2; j++)
    {
        printf("% .6g", io→pointlisti*2+j);
    }
    if (io→numberofpointattributes > 0)
    {
        printf("%attributes");
    }
    for (j = 0; j < io→numberofpointattributes; j++)
    {
        printf("% .6g", io→pointattributelisti*io→numberofpointattributes+j);
    }
    if (markers)
    {
        printf("%marker %d\n", io→pointmarkerlisti);
    }
    else
    {
        printf("\n");
    }
}
printf("\n");
if (reporttriangles ∨ reportneighbors)
{
    for (i = 0; i < io→numberoftriangles; i++)
    {
        if (reporttriangles)
        {
            printf("Triangle %4d points:", i);
            for (j = 0; j < io→numberofcorners; j++)
            {

```

```

        printf(" %4d", io->trianglelist[i*io->numberofcorners+j]);
    }
if (io->numberoftriangleattributes > 0)
{
    printf(" attributes");
}
for (j = 0; j < io->numberoftriangleattributes; j++)
{
    printf(" %.6g", io->triangleattributelist[i*io->numberoftriangleattributes+j]);
}
printf("\n");
if (reportneighbors)
{
    printf("Triangle %4d neighbors:", i);
    for (j = 0; j < 3; j++)
    {
        printf(" %4d", io->neighborlist[i*3+j]);
    }
    printf("\n");
}
printf("\n");
if (reportsegments)
{
    for (i = 0; i < io->numberofsegments; i++)
    {
        printf("Segment %4d points:", i);
        for (j = 0; j < 2; j++)
        {
            printf(" %4d", io->segmentlist[i*2+j]);
        }
        if (markers)
        {
            printf(" marker %d\n", io->segmentmarkerlist[i]);
        }
        else
        {
            printf("\n");
        }
    }
    printf("\n");
}
if (reportededges)
{
    for (i = 0; i < io->numberofedges; i++)
    {
        printf("Edge %4d points:", i);
        for (j = 0; j < 2; j++)
        {

```

```

printf(" %4d", io→edgelisti*2+j);
}
if (reportnorms ∧ (io→edgelisti*2+1 ≡ -1))
{
  for (j = 0; j < 2; j++)
  {
    printf(" %.6g", io→normlisti*2+j);
  }
}
if (markers)
{
  printf("marker %d\n", io→edgemarkerlisti);
}
else
{
  printf("\n");
}
printf("\n");
}
if (reportholes)
{
  for (i = 0; i < io→numberofholes; i++)
  {
    printf("Hole %4d points:", i);
    for (j = 0; j < 2; j++)
    {
      printf(" %.6g", io→holelisti*2+j);
    }
  }
  printf("\n");
}
}

```

See also section 37.

This code is used in section 7.1.

## 37 C interface routine to Triangle

This is based loosely on the *tricall* example that is provided with the Triangle package.

```
(C Functions 36)C +≡
void poly2triangles_(npoints, polygon, area, nholes, hole, refine, ntriangles, triangles, markers)
{
    int *npoints; // Input
    int *nholes;
    int *refine;
    double *area;
    double (*polygon)2;
    double (*hole)2;

    int *ntriangles; // Output
    double (*triangles)4,2;
    int (*markers)4;
}

struct triangulateio in, mid, out; // Local
struct triangulateio *final;
int i, j, n, nh, pt, jt, k, idup, nunq, ip, munq;
int pointmapg2-num_points, pointinvmapg2-num_points;
int segmapg2-num_points,2;

n = *npoints; /* Have rewritten this code to eliminate duplicate points and segments from the
   input polygon. */
nunq = 0; // Number of unique points
for (i = 0; i < n; ++i)
{
    idup = 0; // = 0 for a new point
    if (i > 0)
    {
        for (j = 0; j < i; ++j)
        {
            if ((polygoni,0 ≡ polygonj,0) ∧ (polygoni,1 ≡ polygonj,1))
            {
                pointmapi = pointmapj; // A duplicate point;
                idup = 1; // keep track of which it matched
            }
        }
    }
    if (idup ≡ 0)
    {
        pointmapi = nunq; // Unique point number as fn. of original
        pointinvmapnunq = i; // The inverse of that
        nunq++; // Increment after =i arrays start at 0
    }
} /* Analogous code for segments. Note that the orientation of the segments is unimportant to
   Triangle, so we must test both directions in identifying duplicates. */
munq = 0; // Number of unique segments
for (i = 0; i < n; ++i)
{
    /* We need to explicitly close our polygon by setting up a segment from the last point to the
       first point. Can do that most transparently by defining ip as: */
    if (i ≡ n - 1)
```

```

{
    ip = 0;
}
else
{
    ip = i + 1;
}
if (pointmapi ≡ pointmapip)
{
    idup = 1; // Throw out trivial segments
}
else
{
    idup = 0;
    if (munq > 0)
    {
        for (j = 0; j < munq; ++j)
        {
            if (((pointmapi ≡ segmapj,0) ∧ (pointmapip ≡ segmapj,1)) ∨ (((pointmapi ≡
                segmapj,1) ∧ (pointmapip ≡ segmapj,0))))
            {
                idup = 1; // A genuine duplicate segment
            }
        }
    }
}
if (idup ≡ 0)
{
    // Define map for unique segments
    segmapmunq,0 = pointmapi; // 0 and 1 just denote the ends
    segmapmunq,1 = pointmapip;
    munq++; // Again, so arrays start at 0
}
/* Can now allocate and set arrays that will be passed to Triangle. */
in.numberofpoints = nunq;
in.numberofpointattributes = 0;
in.pointlist = (double *) malloc(in.numberofpoints * 2 * sizeof(double));
in.pointmarkerlist = (int *) malloc(in.numberofpoints * sizeof(int));
j = 0;
for (i = 0; i < nunq; ++i)
{
    in.pointlistj = polygonpointinvmapi,0;
    j++;
    in.pointlistj = polygonpointinvmapi,1;
    j++; /* Point markers are intended to be assigned in a manner analogous to that used above
           for breakup-polygon. However, Triangle only respects nonzero markers. So, add 1 here and
           subtract when transferring back to main code. */
    in.pointmarkerlisti = pointinvmapi + 1;
}
in.numberofsegments = munq;
in.segmentlist = (int *) malloc(in.numberofsegments * 2 * sizeof(int));
j = 0;
for (i = 0; i < munq; ++i)

```

```

{
  in.segmentlistj = segmapi,0;
  j++;
  in.segmentlistj = segmapi,1;
  j++;
}

nh = *nholes;
in.numberofholes = nh;
if (nh > 0)
{
  in.holelist = (double *) malloc(in.numberofholes * 2 * sizeof(double));
  j = -1;
  for (i = 0; i < nh; ++i)
  {
    j++;
    in.holelistj = holei,0;
    j++;
    in.holelistj = holei,1;
  }
}
else
{
  in.holelist = (double *) NULL;
}
in.numberoftregions = 0;
in.numberoftriangles = 0;
in.regionlist = (double *) NULL;
in.segmentmarkerlist = (int *) NULL;
@if 0
printf("Input_point_set:\n\n");
report(&in, 0, 0, 0, 0, 0, 0, 1);
#endif
mid.pointlist = (double *) NULL;
mid.pointmarkerlist = (int *) NULL;
mid.trianglelist = (int *) NULL;
mid.segmentlist = (int *) NULL;
mid.segmentmarkerlist = (int *) NULL;
mid.edgelist = (int *) NULL;
mid.edgemarkerlist = (int *) NULL;
mid.normlist = (double *) NULL;
triangulate("pzQ", &in, &mid, (struct triangulateio *) NULL);
#if 0
printf("Initial_triangulation:\n\n");
report(&mid, 0, 1, 0, 1, 0, 0, 1);
#endif
out.pointlist = (double *) NULL;
out.pointmarkerlist = (int *) NULL;
out.trianglelist = (int *) NULL;
out.segmentlist = (int *) NULL;
out.segmentmarkerlist = (int *) NULL;
out.edgelist = (int *) NULL;

```

```

out.edgemarkerset = (int *) NULL;
out.normlist = (double *) NULL;

if (*refine == TRUE)
{
    char args1[16] = "rpqa";
    char args2 = "zYQ";
    char areastr[9];

    sprintf(areastr, "%8.6f", *area);
    strcat(args1, areastr);
    strcat(args1, args2);

@#if 0
    The minimum area here should probably be user specified.
    triangulate("rpqzYQ", &mid, &out, (struct
        triangulateio *) NULL);
@#endif
    triangulate(args1, &mid, &out, (struct triangulateio *) NULL);

@#if 0
    printf("Final triangulation:\n\n");
    report(&out, 0, 1, 0, 1, 0, 0, 1);
@#endif

    final = &out;
}
else
{
    final = &mid;
}

if (final->numberofcorners != 3)
{
    printf("STOP!!! Triangle thinks triangles have %i corners",
        final->numberofcorners);
}
if (final->numberoftangles > max_triangles - 1)
{
    printf("STOP!!! Number of triangles %i exceeds dimensions of triangles array!",
        final->numberoftangles);
}

for (i = 0; i < final->numberoftangles; ++i)
{
    for (j = 0; j < final->numberofcorners; ++j)
    {
        pt = final->trianglelist[i*final->numberofcorners+j];
        jt = final->numberofcorners - j; // Reverse order for decompose_polygon
        triangles[i,jt,0] = final->pointlist[2*pt];
        triangles[i,jt,1] = final->pointlist[2*pt+1];
        /* Subtract 1 here to undo the shift made prior to the first call. */
        markers[i,jt] = final->pointmarkerlist[pt] - 1;
    }
    triangles[i,0,0] = triangles[i,3,0];
    triangles[i,0,1] = triangles[i,3,1];
    markers[i,0] = markers[i,3];
}
*ntriangles = final->numberoftangles;

```

```
free(in.pointlist);
free(in.segmentlist);
free(in.pointmarkerlist);
free(in.regionlist);
free(in.segmentmarkerlist);
free(mid.pointlist);
free(mid.segmentlist);
free(mid.pointmarkerlist);
free(mid.trianglelist);
free(mid.segmentmarkerlist);
free(mid.edgelist);
free(mid.edgemarkerlist);
free(mid.normlist);
free(out.pointlist);
free(out.segmentlist);
free(out.pointmarkerlist);
free(out.trianglelist);
free(out.segmentmarkerlist);
free(out.edgelist);
free(out.edgemarkerlist);
free(out.normlist);
if (nh > 0)
{
    free(in.holelist);
}
}
```

## 38 INDEX

*a:* 24, 25.  
*a\_hat:* 25.  
*a\_vec:* 25.  
*a\_vertex:* 25.  
*a\_y:* 9, 9.4, 15.  
*abs:* 9.4, 11, 15, 16, 23, 24.  
*add\_surface:* 24.  
*add\_transform:* 24.  
*alpha:* 25.  
*aname:* 8.  
*apex:* 24.  
*area:* 25, 27, 37<sup>C</sup>.  
*areal:* 9.  
*areastrings:* 37<sup>C</sup>.  
*args1:* 37<sup>C</sup>.  
*args2:* 37<sup>C</sup>.  
*assert:* 9, 9.2, 9.3, 9.4, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 21.1, 21.2, 21.3, 21.4, 21.5, 21.6, 21.7, 21.8, 21.9, 22, 22.1, 23, 24, 25, 27, 29, 30, 31, 32, 34, 35.  
*aunit:* 8.  
*aux\_sector\_ind:* 9.  
*aux\_stratum:* 1, 8, 9, 9.1, 14, 17, 21, 35.  
*aux\_stratum\_def:* 21.  
*aux\_stratum\_points:* 8, 9, 9.1, 14, 35.  
*aux\_stratum\_poly:* 8, 9, 9.1, 14, 35.  
*aux\_stratum\_pts:* 21.  
*aux\_stratum\_segment:* 8, 9, 9.1, 14, 17, 35.  
*aux\_stratum\_start:* 21.  
*aux1\_done:* 35.  
*b:* 9, 20, 21, 22, 24, 25.  
*b\_vec:* 25.  
*b\_vertex:* 25.  
*b\_y:* 9, 9.4, 15.  
*backward\_loop:* 21.  
*bc\_cross:* 25.  
*be:* 37<sup>C</sup>.  
*boltzmanns\_const:* 16, 35.  
*boundaries\_neighbors:* 9.  
*bounds:* 1, 9.4, 19, 21.  
*boxgen:* 9.4.  
*breakup\_polygon:* 1, 8, 14, 21, 37<sup>C</sup>.  
*btopdetector:* 1.  
  
*c:* 25.  
*c\_hat:* 25.  
*c\_perp:* 25.  
*c\_vec:* 25.  
*c\_vertex:* 25.  
  
*center:* 9, 12, 14, 15, 23, 25, 26.  
*CHAR:* 9.  
*char\_undef:* 9, 20, 21.  
*char\_uninit:* 8, 17.  
*check\_geometry:* 9.  
*clear\_polygon:* 1, 8, 14, 21, 21.6.  
*close\_elements:* 33.  
*close\_node:* 33.  
*close\_tips:* 33.  
*closed\_curve:* 21, 21.7.  
*closest\_distance:* 34.  
*closest\_elements:* 33, 34.  
*closest\_node:* 33, 34.  
*closest\_point:* 33, 34.  
*closest\_tips:* 33, 34.  
*coeff:* 9, 9.4, 15, 24.  
*command\_arg:* 8.  
*common\_node:* 21, 21.8.  
*composite:* 24.  
*cone:* 24.  
*const:* 9, 9.1, 9.2, 16, 19, 22.1, 25, 26.  
*corner:* 21, 21.3.  
*corner\_ptr:* 18.  
*cos:* 9.4, 15, 23, 24.  
*cos\_y:* 9, 9.4, 15, 24.  
*count\_curves:* 21.7, 31.  
*csec:* 20.  
*current\_int\_props:* 9, 9.1, 12, 14.  
*current\_real\_props:* 9, 9.1, 12, 14.  
*curve\_closed:* 21, 21.7, 31.  
*curve\_open:* 21, 31.  
*curve\_start:* 21, 21.7, 31.  
*curve\_start\_tip:* 21, 21.7, 31.  
*curve\_type:* 21, 21.7, 31.  
*cylindrical:* 1.  
*cylindrical\_hw:* 1.  
*cylindrical\_section:* 1.  
  
*declare\_varp:* 9.  
*decompose\_polygon:* 1, 12, 14, 15, 37<sup>C</sup>.  
*def\_geom\_2d\_main:* 8, 9.  
*default\_diag\_setup:* 9.  
*define\_dimen:* 9.  
*define\_sector:* 16, 35.  
*define\_sector\_exit:* 35.  
*define\_sector\_plasma:* 16, 35.  
*define\_sector\_target:* 16, 35.  
*define\_sector\_vacuum:* 16, 35.  
*define\_sector\_wall:* 16, 35.  
*define\_surface:* 9.4, 15, 24.  
*define\_surface\_a:* 24.  
*define\_varp:* 9.  
*definegeometry2d:* 1, 8.

*def2ddetector*: 1.  
*degas2*: 1.  
*delta*: 21, 21.9, 34.  
*delta\_p*: 33.  
*den*: 25.  
*detector\_setup*: 1, 9.  
*dg\_elements\_list*: 8, 9, 9.1, 9.4, 10, 14, 21, 21.4.  
*dg\_file*: 1, 21.  
*dg\_loop*: 1, 10.  
*dg\_poly\_ind*: 9.  
*dg\_poly\_meshcon*: 8, 9, 9.1, 9.4, 10, 14, 21, 21.8, 21.9.  
*dg\_poly\_meshcon\_elem\_ind*: 9.  
*dg\_poly\_meshcon\_hv*: 8, 9, 9.1, 9.4, 10, 14, 21, 21.8, 21.9.  
*dg\_poly\_meshcon\_ind*: 9.  
*dg\_poly\_num\_elements*: 8, 9, 9.1, 9.4, 10, 14, 21, 21.4.  
*dg\_poly\_num\_meshcon*: 8, 9, 9.1, 9.4, 10, 14, 21, 21.6.  
*dg\_polygon*: 1, 21.4, 21.6.  
*dg\_polygons*: 8, 9, 9.1, 9.4, 10, 14, 21, 21.4.  
*diag\_grp\_init*: 17.  
*diag\_loop*: 22.  
*diag\_mult*: 8, 9, 9.1, 17.  
*diag\_name*: 8, 9, 9.1, 17.  
*diag\_solid*: 8, 9, 9.1, 17.  
*diag\_spacing*: 8, 9, 9.1, 17.  
*diag\_stratum*: 8, 9, 9.1, 17.  
*diag\_tab\_index*: 8, 9, 9.1, 17.  
*diag\_var\_max*: 8, 9, 9.1, 17.  
*diag\_var\_min*: 8, 9, 9.1, 17.  
*diag\_variable*: 8, 9, 9.1, 17.  
*diags\_ind*: 9.  
*dim\_aux\_sectors*: 8, 9, 9.1.  
*dim\_curves*: 21, 31.  
*dim\_dg\_poly*: 8, 9, 9.1, 9.4.  
*dim\_diags*: 8, 9, 9.1.  
*dim\_elements*: 8, 9, 9.1, 9.4.  
*dim\_nodes*: 8, 9, 9.1, 9.4.  
*dim\_walls*: 8, 9, 9.1, 9.4, 11.  
*dim\_y*: 8, 9, 9.1, 9.4.  
*dim\_ym*: 8, 9, 9.1, 9.4.  
*diskin*: 9, 14, 21.  
*diskin2*: 9, 10, 11.  
*diskout*: 9, 20, 21.  
*dist*: 21, 21.9, 34.  
*dist\_p\_edge*: 33.  
*dummy*: 18.  
*e*: 9, 20, 21, 22.  
*edge*: 1.  
*edge\_chosen*: 33.  
*edge\_elements*: 34.  
*edge\_num\_elements*: 34.  
*edge\_p*: 33.  
*edge\_point*: 34.  
*edge\_reverse*: 21, 21.3.  
*edgelist*: 36<sup>C</sup>, 37<sup>C</sup>.  
*edgemarkerlist*: 36<sup>C</sup>, 37<sup>C</sup>.  
*electron\_charge*: 22.1.  
*element\_assigned*: 8, 9, 9.1, 9.4, 10.  
*element\_curve\_num*: 31.  
*element\_end*: 8, 9, 10, 13, 21, 21.8, 29, 30, 31, 33, 34.  
*element\_ends\_ind*: 9.  
*element\_ind*: 9.  
*element\_missing*: 8, 9, 9.1, 9.4, 10.  
*element\_skipped*: 8, 9, 9.1, 9.4, 10.  
*element\_start*: 8, 9, 10, 13, 21, 21.4, 21.8, 21.9, 29, 30, 31, 33, 34.  
*elements\_list*: 29, 30, 31.  
*elements\_to\_polygon*: 21.4, 21.7, 21.9, 30, 31.  
*end\_aux\_stratum*: 1.  
*end\_detectors*: 9.  
*end\_diagnostic*: 1.  
*end\_elements*: 21, 21.4, 29, 31.  
*end\_faces*: 9, 15.  
*end\_nodes*: 21, 21.4, 29, 31.  
*end\_point*: 21, 21.4.  
*end\_prep*: 1.  
*end\_pt*: 9, 12.  
*end\_sectors*: 9.  
*end\_tip*: 21, 21.4.  
*end\_tips*: 21, 21.4, 29, 31.  
*end\_zones*: 9, 9.1, 15, 16.  
*eof*: 9.  
*epsilon\_angle*: 15, 24.  
*erase\_geometry*: 9.  
*exp\_inc*: 9, 10.  
*exp\_string*: 9, 11.  
*face*: 24.  
*facearray*: 8, 9, 9.1, 9.4, 12, 14, 16.  
*face1*: 9, 16, 24, 35.  
*face2*: 24.  
*FALSE*: 9, 9.1, 10, 11, 13, 14, 16, 17, 21, 21.3, 21.9, 22, 29, 30, 31, 35.  
*file*: 8, 9, 11, 18, 20, 21.  
*FILE*: 1.  
*file\_format*: 9, 20.  
*fileid*: 9.  
*FILELEN*: 8, 9, 20, 21.  
*filename*: 8, 9.  
*final*: 37<sup>C</sup>.  
*find\_endpoints*: 21.4, 29, 31.

*find\_min\_max*: 9, [19](#).  
*find\_poly\_zone*: 16, 35.  
*FLOAT*: 9.  
*form*: 8, 9, 11, 18.  
*forward\_loop*: [21](#).  
*free*: [37<sup>C</sup>](#).  
*gamma*: [25](#).  
*geom\_epsilon*: 9.4, 11.  
*geom\_infinity*: 34.  
*geom\_translate*: 24.  
*geometry*: 1.  
*geometry\_symmetry*: 23, 24.  
*geometry\_symmetry\_cyl\_hw*: 9, 9.2, 9.3, 9.4, 16, 23, 24.  
*geometry\_symmetry\_cyl\_section*: 9, 9.2, 9.3, 9.4, 15, 16, 23, 24.  
*geometry\_symmetry\_cylindrical*: 9, 9.2, 9.3, 9.4, 23, 24.  
*geometry\_symmetry\_none*: 9, 9.1.  
*geometry\_symmetry\_oned*: 9, 9.2, 9.3, 23, 24.  
*geometry\_symmetry\_plane*: 9, 9.2, 9.3, 9.4, 23, 24.  
*geometry\_symmetry\_plane\_hw*: 9, 9.2, 9.3, 9.4, 15, 16, 23, 24.  
*geometry2d*: 14.  
*get\_mc\_edge\_seg*: 21.8, 21.9, [32](#).  
*gi\_common*: 9, 23, 24.  
*gi\_ext*: 9, 23, 24, 35.  
*grp\_sectors*: 9, 17.  
*g2\_common*: 9.  
*g2\_neddecl*: 9.  
*g2\_ncdef*: 9.  
*g2\_ncwrite*: 9.  
*g2\_num\_points*: 8, 9, 10, 20, 21, 21.1, 21.2, 21.3, 21.9, 23, 30, 31, 35, [37<sup>C</sup>](#).  
*g2\_num\_polygons*: 8, 9, 9.1, 12, 14, 16, 17.  
*g2\_points\_ind*: 9.  
*g2\_points\_ind0*: 9.  
*g2\_poly\_ind*: 9.  
*g2\_polygon\_points*: 8, 9, 12, 14.  
*g2\_polygon\_segment*: 8, 9, 12, 14.  
*g2\_polygon\_stratum*: 8, 9, 12, 14, 17.  
*g2\_polygon\_xz*: 8, 9, 12, 14.  
*g2\_polygon\_zone*: 8, 9, 12, 14, 16, 17.  
*g2\_x*: 8, 9, 9.4, 10, 11, 12, 13, 14, 15, 18, 19, 20, 21, 21.1, 21.2, 21.3, 21.4, 21.7, 21.8, 21.9, 23, 25, 26, 27, 28, 30, 33, 34, 35.  
*g2\_xz\_ind*: 9.  
*g2\_z*: 8, 9, 9.4, 10, 11, 12, 13, 14, 15, 18, 19, 20, 21, 21.1, 21.2, 21.3, 21.4, 21.8, 21.9, 23, 25, 26, 27, 28, 30, 33, 34, 35.  
*half*: 23, 24, 25, 27.  
*here*: [37<sup>C</sup>](#).  
*hole*: [37<sup>C</sup>](#).  
*holelist*: [36<sup>C</sup>](#), [37<sup>C</sup>](#).  
*hweb*: 1, 9.  
*i*: 9, 18, [21](#), 24, 27, 28, 29, 30, 31, 33, 34, [36<sup>C</sup>](#), [37<sup>C</sup>](#).  
*i\_aux*: 9, 14, 17, [35](#).  
*i\_diag*: 9, 17.  
*i\_edge*: [32](#).  
*i\_elem*: [32](#).  
*i\_grp*: 9, 17.  
*i\_inc\_g2*: 8, 9.  
*i\_init*: [31](#).  
*i\_mc*: 21, 21.9.  
*i\_poly*: 9, 16, 17, [35](#).  
*i\_prop*: 9, 12, 14, 21.  
*i\_sect*: 9, 16, 17.  
*i\_tot*: 21, 21.7, 21.9, [31](#).  
*i\_2*: 9.  
*i\_2\_loop*: 9.  
*i\_2\_range*: 9.  
*idup*: [37<sup>C</sup>](#).  
*iedge*: 1, 9, 13, 21, 21.4, 21.6, 21.9.  
*iedge\_loop*: [21](#), 21.5.  
*iedge\_order*: 21, 21.5.  
*iedge\_p*: 21, 21.9.  
*iedge\_points*: 21.  
*iedge\_temp*: [21](#), 21.9.  
*iedges*: [21](#), 21.5, 21.7, 21.9, [33](#).  
*iedge1*: [33](#).  
*iedge2*: [33](#).  
*ielement*: 9, 10.  
*ielement2*: 9, 10.  
*iend*: [20](#).  
*implicit\_none\_f77*: 8, 9, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35.  
*implicit\_none\_f90*: 8, 9, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35.  
*in*: 1, [37<sup>C</sup>](#).  
*inc\_pt*: 9, 12.  
*increment\_g2\_num\_polygons*: 8, 12, 14.  
*increment\_num\_aux\_sectors*: 8, 14.  
*increment\_num\_dg\_poly*: 8, 10.  
*increment\_num\_diags*: 8, 9.  
*increment\_num\_elements*: 8, 10.  
*increment\_num\_nodes*: 8, 10, 11.  
*increment\_num\_walls*: 8, 10.  
*increment\_y\_div*: 8, 9.  
*ind\_x*: [23](#).  
*ind\_z*: [23](#).  
*index*: 10.  
*init\_geometry*: 9.4.  
*init\_identity*: 24.

*initial\_element*: 30.  
*initial\_tip*: 30.  
*inode*: 9, 10, 11, 13, 19, 20.  
*INT*: 9.  
*int\_props*: 21.  
*int\_undef*: 9.1, 10, 16, 21, 21.6, 21.7, 32.  
*int\_uninit*: 8, 14, 21, 22.  
*int\_unused*: 8, 9.1, 9.4, 10, 16.  
*invert*: 24.  
*io*: 36<sup>C</sup>.  
*iostat*: 8.  
*ip*: 37<sup>C</sup>.  
*is\_aux\_sector*: 9, 17.  
*isec*: 20.  
*iseg*: 9, 10, 11, 13, 20.  
*istart*: 20.  
*itip*: 9, 13.  
*iwall*: 9, 10, 11, 20.  
*ix*: 8, 9, 12, 13, 14, 15, 18, 21, 21.3.  
*ix\_max*: 9, 13.  
*ix\_min*: 9, 13.  
*ix\_start*: 21, 21.3.  
*ix\_step*: 21, 21.3.  
*ix\_stop*: 21, 21.3.  
*ixi*: 21, 21.3.  
*ixlb*: 18.  
*ixmdp*: 18.  
*ixpt1*: 18.  
*ixpt2*: 18.  
*ixrb*: 18.  
*iy*: 23.  
*iysptrx1*: 18.  
*iysptrx2*: 18.  
*iz*: 8, 9, 12, 13, 14, 15, 18, 21, 21.3.  
*iz\_max*: 9, 13.  
*iz\_min*: 9, 13.  
*iz\_start*: 21, 21.3.  
*iz\_step*: 21, 21.3.  
*iz\_stop*: 21, 21.3.  
*izi*: 21, 21.3.  
*j*: 9, 29, 30, 31, 33, 34, 35, 36<sup>C</sup>, 37<sup>C</sup>.  
*j\_fn*: 21, 21.9.  
*j\_init*: 21, 21.9.  
*j\_match*: 30, 31.  
*j\_max*: 21, 21.9.  
*j\_min*: 21, 21.9.  
*j\_seq*: 32.  
*jt*: 37<sup>C</sup>.  
*jxz*: 8.  
*k*: 9, 25, 37<sup>C</sup>.  
*k\_zone1*: 9, 16, 35.  
*k\_zone2*: 9, 16, 35.  
*keyword*: 9, 21, 22.  
*lc*: 25.  
*len*: 9, 10, 11, 21, 22.  
*length*: 9, 10, 11, 21, 22.  
*line*: 8, 9, 9.2, 9.3, 10, 11, 21, 21.1, 21.2, 21.3, 21.4, 21.5, 22, 22.1.  
*LINELEN*: 9, 20, 21, 22.  
*lookup\_surface*: 35.  
*loop1*: 9.  
*m\_test1*: 25.  
*m\_test2*: 25.  
*ma\_check*: 9, 21.  
*ma\_common*: 9, 21.  
*ma\_lookup*: 9, 21.  
*malloc*: 37<sup>C</sup>.  
*markers*: 36<sup>C</sup>, 37<sup>C</sup>.  
*mass\_in*: 17.  
*match*: 29, 30, 31.  
*match\_edge\_to\_poly*: 33.  
*material*: 1.  
*materials\_infile*: 1.  
*max*: 8, 9, 11, 16, 19, 20.  
*max\_corner*: 9, 9.4, 15, 24.  
*max\_grp\_sectors*: 8, 9, 17.  
*max\_segs*: 20.  
*max\_triangles*: 8, 9, 14, 37<sup>C</sup>.  
*max\_xpts*: 18.  
*maximum*: 1.  
*mc\_edge*: 21, 21.8, 21.9, 32.  
*mc\_node*: 21, 21.8.  
*mc\_seg*: 21, 21.8, 21.9, 32.  
*mem\_inc*: 9.1.  
*mesh\_corner*: 9, 13.  
*mesh\_corner\_ind*: 9.  
*mesh\_curve\_num*: 9, 9.4, 14, 21, 21.7.  
*mesh\_east*: 8, 13, 21.5.  
*mesh\_edge\_dg\_label*: 9, 9.4, 13, 14, 21, 21.8, 21.9, 32.  
*mesh\_edge\_elements*: 9, 9.4, 13, 14, 21, 21.7, 21.8, 21.9, 33.  
*mesh\_edge\_elements\_ind*: 9.  
*mesh\_edge\_hv*: 9, 9.4, 13, 14, 21, 21.8, 21.9, 32.  
*mesh\_edge\_num\_elements*: 9, 13, 14, 21, 21.7, 21.8, 21.9, 32, 33.  
*mesh\_elem*: 9, 10.  
*mesh\_elements*: 9, 9.4, 13, 14, 21, 21.7, 21.8, 21.9, 33, 34.  
*mesh\_elements\_ind*: 9.  
*mesh\_h*: 8, 10, 13, 21.  
*mesh\_ix\_end*: 9, 13.

*mesh\_ix\_start*: 9, 13.  
*mesh\_ix\_step*: 9, 13.  
*mesh\_iz\_end*: 9, 13.  
*mesh\_iz\_start*: 9, 13.  
*mesh\_iz\_step*: 9, 13.  
*mesh\_nodes*: 9, 9.4, 13, 14, 21, 21.7, 21.8, 21.9, 33, 34.  
*mesh\_nodes\_ind*: 9.  
*mesh\_north*: 8, 13, 21.5.  
*mesh\_num\_edges*: 8, 9, 13, 21, 21.5, 21.9, 32, 33.  
*mesh\_pt*: 9, 12.  
*mesh\_scratch*: 9, 9.4, 14, 21, 21.7, 21.9, 33.  
*mesh\_sense*: 9, 12.  
*mesh\_senw\_ind*: 9.  
*mesh\_south*: 8, 13, 21.5.  
*mesh\_tot\_elements*: 9, 13.  
*mesh\_tot\_nodes*: 9, 13.  
*mesh\_v*: 8, 10, 13, 21.  
*mesh\_west*: 8, 13, 21.5.  
*mesh\_x*: 18.  
*mesh\_xz*: 8, 9, 9.4, 14, 18, 21, 21.3.  
*mesh\_xz\_ind*: 9.  
*mesh\_xzm*: 8, 12, 13.  
*mesh\_z*: 18.  
*meshcon*: 32.  
*meshcon\_elem\_max*: 8, 9, 10, 21, 32.  
*meshcon\_hv*: 32.  
*meshcon\_loop*: 1, 10.  
*meshcon\_max*: 8, 9, 10, 21.  
*mid*: 37<sup>C</sup>.  
*min*: 19, 20, 21.9.  
*min\_corner*: 9, 9.4, 15, 24.  
*min\_dist*: 21, 21.9.  
*minimum*: 1, 37<sup>C</sup>.  
*miss\_elem*: 9, 10.  
*mixed\_type\_loop*: 1, 9.  
*mod*: 14, 21.5, 21.9.  
*mult*: 9, 10, 22, 22.1.  
*multiplier*: 1.  
*munq*: 37<sup>C</sup>.  
*n*: 9, 21, 23, 27, 28, 29, 30, 31, 37<sup>C</sup>.  
*n\_iedge*: 21, 21.9.  
*nc*: 9.  
*NC\_CLOBBER*: 9.  
*nc\_decls*: 9.  
*nc\_read\_materials*: 8.  
*nc\_stat*: 9.  
*ncclose*: 9.  
*nccreate*: 9.  
*ncendef*: 9.  
*NCOEFFS*: 9, 24.  
*neg\_y*: 25.  
*neighborlist*: 36<sup>C</sup>.  
*new\_diagnostic*: 1.  
*new\_node*: 9, 13.  
*new\_other\_sector*: 9, 16.  
*new\_polygon*: 1.  
*new\_pt*: 9, 12.  
*new\_solid\_sector*: 9, 16.  
*new\_zone*: 1.  
*new\_zone\_type*: 9, 14.  
*new\_zone\_type\_2*: 9.  
*next\_surface*: 1.  
*next\_token*: 9, 9.2, 9.3, 10, 11, 21, 21.1, 21.2, 21.3, 21.4, 21.5, 22, 22.1.  
*nh*: 37<sup>C</sup>.  
*nholes*: 37<sup>C</sup>.  
*node*: 30, 31.  
*node\_element\_count*: 8, 9, 9.1, 9.4, 10.  
*node\_ind*: 9.  
*node\_many\_elements*: 8, 10.  
*node\_mixed\_normals*: 8, 10.  
*node\_no\_elements*: 8, 10, 19.  
*node\_one\_element*: 8, 10.  
*node\_regular*: 8, 10.  
*node\_type*: 8, 9, 9.1, 9.4, 10, 19.  
*node\_undefined*: 8.  
*nodes*: 8, 9, 9.1, 9.4, 10, 11, 14, 19, 20, 21, 21.2, 21.4, 30.  
*normlist*: 36<sup>C</sup>, 37<sup>C</sup>.  
*nout*: 21.  
*npoints*: 37<sup>C</sup>.  
*nsectors*: 16, 17.  
*nt*: 9, 14.  
*ntriangles*: 9, 14, 37<sup>C</sup>.  
*NULL*: 37<sup>C</sup>.  
*num*: 21, 21.3.  
*num\_aux\_sectors*: 8, 9, 9.1, 14, 17, 35.  
*num\_aux\_stratum\_pts*: 21.  
*num\_closest*: 34.  
*num\_curves*: 21, 21.7, 31.  
*num\_dg\_poly*: 8, 9, 9.1, 9.4, 10, 14, 21, 21.4.  
*num\_diags*: 8, 9, 9.1, 17.  
*num\_elements*: 8, 9, 9.1, 9.4, 10.  
*num\_ends*: 21, 21.4, 29, 31.  
*num\_h\_elems*: 9, 10.  
*num\_iedges*: 21, 21.5, 21.6, 21.7, 21.9.  
*num\_mesh\_elems*: 9, 10.  
*num\_new\_walls*: 1, 9, 11.  
*num\_nodes*: 8, 9, 9.1, 9.4, 10, 11, 19.  
*num\_polygons*: 35.  
*num\_sections*: 20.  
*num\_walls*: 8, 9, 9.1, 9.4, 10, 11, 14, 20, 21, 21.2.  
*num\_y*: 9.

*num\_zone1*: 9, 16, 35.  
*num\_zone2*: 9, 16, 35.  
*number*: 1.  
*numberofcorners*: 36<sup>C</sup>, 37<sup>C</sup>.  
*numberofedges*: 36<sup>C</sup>.  
*numberofholes*: 36<sup>C</sup>, 37<sup>C</sup>.  
*numberofpointattributes*: 36<sup>C</sup>, 37<sup>C</sup>.  
*numberofpoints*: 36<sup>C</sup>, 37<sup>C</sup>.  
*numberofregions*: 37<sup>C</sup>.  
*numberofsegments*: 36<sup>C</sup>, 37<sup>C</sup>.  
*numberoftriangleattributes*: 36<sup>C</sup>.  
*numberoftriangles*: 36<sup>C</sup>, 37<sup>C</sup>.  
*nunit*: 9, 18, 20, 21, 22.  
*nunq*: 37<sup>C</sup>.  
*nx\_nz\_max*: 9, 9.4.  
*nxd*: 8, 9, 9.1, 9.4, 12, 13, 14, 18, 21, 21.3.  
*nxd\_0*: 9, 13.  
*nxpt*: 9, 18.  
*nzd*: 9, 9.1, 9.4, 12, 13, 14, 18, 21, 21.3.  
*nzd\_0*: 9, 13.  
*one*: 9, 9.1, 9.2, 9.4, 12, 15, 16, 21, 22.1, 23, 24, 25.  
*one\_zone\_type*: 9, 14, 15.  
*oned*: 1.  
*open\_file*: 8, 9, 10.  
*open\_stat*: 8, 9.  
*order\_decreasing*: 21, 21.5.  
*order\_increasing*: 21, 21.5.  
*order\_undefined*: 21, 21.5.  
*other\_face*: 9, 16.  
*other\_sector*: 9, 16.  
*other\_seg*: 9, 16.  
*other\_stratum*: 9, 16.  
*other\_type*: 9, 16.  
*other\_zone*: 9, 16.  
*out*: 37<sup>C</sup>.  
*outer*: 1.  
*outer\_loop*: 21, 21.1.  
*p*: 9, 21, 22.  
*parse\_string*: 9, 10, 11, 21, 22.  
*phi\_mid*: 23.  
*PI*: 9, 9.2, 9.4, 15, 22.1, 24.  
*pixel\_map\_test*: 9.  
*plane*: 1, 9.4, 15, 24.  
*plane\_hw*: 1.  
*planea*: 24.  
*point\_loop*: 30, 31.  
*pointattributeclist*: 36<sup>C</sup>.  
*pointinvmmap*: 37<sup>C</sup>.  
*pointlist*: 36<sup>C</sup>, 37<sup>C</sup>.  
*pointmap*: 37<sup>C</sup>.  
*pointmarkerlist*: 36<sup>C</sup>, 37<sup>C</sup>.  
*poly*: 21, 21.4, 21.6, 21.7, 21.8, 21.9.  
*poly\_area*: 21, 21.4, 21.7, 21.8, 21.9.  
*poly\_elem*: 9, 10.  
*poly\_elements*: 29, 30, 31.  
*poly\_hole\_x*: 8, 9.1, 14, 21.  
*poly\_hole\_z*: 8, 9.1, 14, 21.  
*poly\_int\_ind*: 9.  
*poly\_int\_max*: 8, 9, 12, 14, 21, 35.  
*poly\_int\_props*: 8, 9, 12, 14, 16, 35.  
*poly\_loop*: 21.  
*poly\_material*: 8, 9.1, 16, 21, 35.  
*poly\_min\_area*: 8, 9.1, 14, 21.  
*poly\_num\_holes*: 8, 9.1, 14, 21.  
*poly\_num\_points*: 30.  
*poly\_p*: 21, 21.9, 33, 34.  
*poly\_pt*: 9, 12.  
*poly\_real\_ind*: 9.  
*poly\_real\_max*: 8, 9, 12, 14, 21, 35.  
*poly\_real\_props*: 8, 9, 12, 14, 16, 35.  
*poly\_recyc\_coef*: 8, 9.1, 16, 21, 35.  
*poly\_stratum*: 8, 9.1, 14, 16, 21, 35.  
*poly\_temperature*: 8, 9.1, 16, 21, 35.  
*poly\_to\_tri*: 1.  
*poly\_x*: 30.  
*Polygon*: 9, 21, 21.1, 21.2, 21.3, 21.4, 21.7, 21.8,  
 21.9, 23, 37<sup>C</sup>.  
*Polygon\_area*: 21, 21.4, 21.7, 21.8, 21.9, 27.  
*Polygon\_nc\_file*: 1, 9, 9.1.  
*Polygon\_points*: 35.  
*Polygon\_segment*: 35.  
*Polygon\_volume*: 23, 27.  
*Polygon\_xz*: 35.  
*Polygon\_zone*: 35.  
*Polyoutfile*: 21.  
*Poly2triangles*: 14.  
*Poly2triangles\_*: 37<sup>C</sup>.  
*Poly4*: 9, 15.  
*prep\_done*: 9, 9.1, 9.4, 14.  
*print\_min\_max*: 1.  
*print\_polygon*: 1, 21.  
*print\_walls*: 1, 9, 11, 20.  
*printf*: 36<sup>C</sup>, 37<sup>C</sup>.  
*probably*: 37<sup>C</sup>.  
*process\_polygon*: 9, 14, 21, 21.5, 21.6, 35.  
*process\_polygon\_cylindrical*: 15.  
*process\_polygon\_plane*: 23.  
*pt*: 37<sup>C</sup>.  
*quad*: 26.  
*quad\_center*: 12, 14, 15, 26.  
*quit*: 1.  
*read\_int\_soft\_fail*: 10, 11.

*read\_integer*: 9, 10, 21, 21.1, 21.2, 21.3, 21.4, 22.  
*read\_real*: 9, 9.2, 21, 22.  
*read\_real\_scaled*: 10.  
*read\_real\_soft\_fail*: 9, 11.  
*read\_sonnet\_mesh*: 1, 9, 11.  
*read\_string*: 9, 10, 11, 21, 22.  
*read\_uedge\_mesh*: 1, 9, 18.  
*readfilenames*: 8.  
*readgeometry*: 1, 11, 35.  
**REAL**: 7.1<sup>C</sup>.  
*real\_props*: 21.  
*real\_undef*: 9, 9.1, 10, 11, 23.  
*real\_uninit*: 8, 14, 21.  
*real\_unused*: 9.1.  
*recyc\_coef*: 1.  
*refine*: 9, 14, 37<sup>C</sup>.  
*regionlist*: 37<sup>C</sup>.  
*report*: 36<sup>C</sup>, 37<sup>C</sup>.  
*reportedges*: 36<sup>C</sup>.  
*reportholes*: 36<sup>C</sup>.  
*reportneighbors*: 36<sup>C</sup>.  
*reportnorms*: 36<sup>C</sup>.  
*reportsegments*: 36<sup>C</sup>.  
*reporttriangles*: 36<sup>C</sup>.  
*reverse*: 1, 21.3.  
*reverse\_polygon*: 21, 21.4, 21.7, 21.8, 21.9, 28.  
*rtest*: 9.  
  
*sc\_common*: 9, 35.  
*sc\_compare*: 16.  
*sc\_diag\_angle*: 22.1.  
*sc\_diag\_energy*: 22.1.  
*sc\_diag\_name\_string*: 9.  
*sc\_diag\_spacing\_linear*: 22.  
*sc\_diag\_spacing\_log*: 22.  
*sc\_diag\_spacing\_unknown*: 22.  
*sc\_diag\_unknown*: 22.  
*sc\_exit*: 17.  
*sc\_exit\_check*: 17.  
*sc\_plasma*: 17.  
*sc\_plasma\_check*: 17.  
*sc\_target*: 17.  
*sc\_target\_check*: 17.  
*sc\_vacuum*: 17.  
*sc\_vacuum\_check*: 17.  
*sc\_wall*: 17.  
*sc\_wall\_check*: 17.  
*sec\_dg\_meshcon1*: 8, 10.  
*sec\_dg\_meshcon2*: 8, 10.  
*sec\_dg\_poly*: 8, 10.  
*sec\_dg\_wall*: 8, 10.  
*sec\_done*: 8, 10.  
*sec\_miss*: 8, 10.  
  
*sec\_p1*: 8, 10.  
*sec\_p2*: 8, 10.  
*sec\_skip*: 8, 10.  
*sec\_undef*: 8, 10.  
*sect\_zone1*: 9, 16, 35.  
*sect\_zone2*: 9, 16, 35.  
*section*: 9, 10.  
*sector*: 1, 9.  
*sector\_break*: 1, 14.  
*sector\_break2*: 1, 14.  
*sector\_strata\_segment*: 16, 17.  
*sector\_type\_pointer*: 17.  
*sector\_zone*: 17.  
*sector1*: 35.  
*segmap*: 37<sup>C</sup>.  
*segmentlist*: 36<sup>C</sup>, 37<sup>C</sup>.  
*segmentmarkerlist*: 36<sup>C</sup>, 37<sup>C</sup>.  
*set\_zn\_min\_max*: 23.  
*setup\_sectors*: 9, 14, 16, 35.  
*should*: 37<sup>C</sup>.  
*side1\_done*: 35.  
*sin*: 9.4, 15, 23, 24.  
*sin\_y*: 9, 9.4, 15, 24.  
*skip\_elem*: 9, 10.  
*solid*: 22.  
*solid\_face*: 9, 16.  
*solid\_faces*: 9, 9.4, 14, 15.  
*solid\_sector*: 9, 16.  
*solid\_ys*: 9, 9.4, 14, 15.  
*solid\_zone*: 9, 14, 15, 16.  
*solid\_zone\_p*: 9, 16.  
*sonnet\_mesh*: 1.  
*SP*: 18, 20.  
*spacing*: 1, 22.  
*specified*: 37<sup>C</sup>.  
*specify\_diagnostic*: 9, 22.  
*specify\_polygon*: 14, 21, 31.  
*sprintf*: 37<sup>C</sup>.  
*st\_decls*: 9, 20, 21, 22.  
*start*: 9, 10, 21, 21.2.  
*start\_pt*: 9, 12.  
*status*: 8, 9, 11, 18, 20, 21.  
*stderr*: 8, 20, 21, 21.5, 22, 22.1, 23.  
*stdout*: 9, 21.  
*step*: 21, 21.2.  
*strata*: 16, 17.  
*strata\_segment*: 1.  
*stratum*: 1, 22.  
*streat*: 37<sup>C</sup>.  
*string*: 9.  
*surf\_type*: 24.  
*surface\_points*: 24.

*swap\_edges*: 21, 21.9.  
*symmetry*: 1, 9, 9.1, 9.2, 9.3, 9.4, 15, 16, 24.  
*tab\_index*: 22.  
*table\_size*: 20.  
*temp*: 21, 21.2, 21.3.  
*temp\_aux\_segment*: 9, 14.  
*temp\_aux\_stratum*: 9, 14.  
*temp\_holes*: 9, 14.  
*temp\_int\_props*: 9, 14.  
*temp\_num\_stratum\_pts*: 9, 14.  
*temp\_polygon*: 9, 14.  
*temp\_real\_props*: 9, 14.  
*temp\_segment*: 9, 14.  
*temp\_stratum\_pts*: 9, 14.  
*temp\_triangles*: 9, 14.  
*temp\_x*: 9, 10.  
*temp\_z*: 9, 10.  
*temperature*: 1.  
*test\_node*: 9, 13, 29.  
*test\_vec\_1*: 9, 12.  
*test\_vec\_2*: 9, 12.  
*test\_vec\_3*: 9, 12.  
*The*: 37<sup>C</sup>.  
*this\_mat*: 9, 16.  
*this\_node*: 9, 11, 13.  
*this\_poly*: 9, 16.  
*this\_rc*: 9, 16.  
*this\_seg*: 9, 16.  
*this\_stratum*: 9, 16.  
*this\_temp*: 9, 16.  
*tip*: 29, 30, 31.  
*tip\_match*: 30, 31.  
*tmp\_edge*: 21, 21.9.  
*tmp\_seg*: 21, 21.9.  
*tmpfilename*: 9, 10.  
*tri\_to\_sonnet*: 1, 11.  
*triangle*: 25.  
*triangle\_area*: 1.  
*triangle\_centroid*: 12, 14, 25.  
*triangle\_hole*: 1.  
*triangleattributelist*: 36<sup>C</sup>.  
*trianglelist*: 36<sup>C</sup>, 37<sup>C</sup>.  
*triangles*: 37<sup>C</sup>.  
*triangulate*: 37<sup>C</sup>.  
*triangulate\_polygon*: 1, 8, 14, 21.  
*triangulate\_to\_zones*: 1, 8, 14, 21.  
*triangulateio*: 36<sup>C</sup>, 37<sup>C</sup>.  
*tricall*: 36, 37.  
*trim*: 9, 20, 21.  
*TRUE*: 9, 9.1, 9.4, 10, 11, 13, 14, 16, 17, 21, 21.3,  
 21.9, 22, 29, 30, 31, 35, 37<sup>C</sup>.  
*two*: 9, 9.4, 25.  
*tx*: 24.  
*ucd\_plot*: 1.  
*uedge\_mesh*: 1.  
*uniform\_ys*: 1.  
*unit*: 8, 9, 10, 11, 18, 20, 21.  
*universal\_cell\_max*: 23, 24.  
*universal\_cell\_min*: 23, 24.  
*universal\_cell\_vol*: 24.  
*universal\_cell\_3d*: 9.4, 24.  
*update\_zone\_info*: 12, 14, 15, 23.  
*user*: 37<sup>C</sup>.  
*usr2ddetector*: 1.  
*var\_alloc*: 9, 9.1, 9.4.  
*var\_free*: 9.  
*var\_max*: 22.  
*var\_min*: 22.  
*var\_realloca*: 8, 9.  
*var\_reallocb*: 9, 9.4, 10.  
*var\_reallocc*: 11.  
*variable*: 1, 22, 22.1.  
*vc\_abs*: 21.9, 25, 33, 34.  
*vc\_args*: 33.  
*vc\_copy*: 23, 24.  
*vc\_cross*: 12, 25.  
*vc\_decl*: 9, 21, 23, 25, 26, 33, 34, 35.  
*vc\_decls*: 9, 21, 23, 25.  
*vc\_difference*: 21.9, 25, 33, 34.  
*vc\_dummy*: 33, 34.  
*vc\_product*: 12, 25.  
*vc\_set*: 9.4, 12, 14, 15, 21.9, 24, 25, 26, 33, 34, 35.  
*vc\_unit*: 25.  
*vc\_xvt*: 25.  
*vector\_compare*: 24.  
*vol*: 9, 9.4, 24.  
*wall*: 1, 21, 21.2.  
*wall\_break*: 1, 11.  
*wall\_chunk*: 20.  
*wall\_elements*: 8, 9, 9.1, 9.4, 10, 11.  
*wall\_ind*: 9.  
*wall\_line*: 20.  
*wall\_loop*: 1, 10.  
*wall\_loop2*: 1, 10.  
*wall\_nodes*: 8, 9, 9.1, 9.4, 10, 11, 14, 20, 21, 21.2.  
*wall\_segment\_count*: 8, 9, 9.1, 9.4, 10, 11, 14, 20,  
21, 21.2.  
*wallfile*: 1, 21.  
*wallfiles*: 1.  
*wallinfile*: 9, 11.  
*walllfile*: 1.  
*walloutfile*: 9, 20.  
*wall2d\_ind*: 9.

*web*: 1, 24, 35.  
*with\_sonnet*: 9, 11.  
*write\_geometry*: 9.  
*write\_poly\_nc*: 9, 9.1.  
*x*: 24, 27, 28.  
*x\_max*: 9, 19.  
*x\_min*: 9, 19.  
*x\_temp*: 28.  
*x\_wall*: 9, 11.  
*xb\_max*: 9, 9.1, 9.2, 9.4, 14, 21, 21.1.  
*xb\_min*: 9, 9.1, 9.2, 9.4, 14, 21, 21.1.  
*xcut*: 1, 21, 21.3.  
*xz\_tmp*: 9, 11.  
*x0*: 24.  
*x1*: 24, 35.  
*x2*: 24, 35.  
*x3*: 24.  
  
*y\_border*: 9.  
*y\_div*: 8, 9, 9.1, 9.4, 12, 14, 15, 16, 23, 35.  
*y\_frac*: 23.  
*y\_ind*: 9.  
*y\_indm*: 9.  
*y\_loop*: 1, 9.  
*y\_loop2*: 1, 9.  
*y\_mat*: 9, 9.1, 16.  
*y\_max*: 9, 16.  
*y\_max\_zone*: 1.  
*y\_min\_zone*: 1.  
*y\_p\_stratum*: 9, 9.1, 16.  
*y\_sect\_ind*: 9.  
*y\_stratum*: 9, 9.1, 16.  
*y\_values*: 1, 8, 9, 9.1, 9.4, 12, 14, 15, 23.  
*yb*: 9.2.  
*yb\_max*: 9, 9.1, 9.2, 9.4.  
*yb\_min*: 9, 9.1, 9.2, 9.4.  
*yhat*: 9, 12.  
  
*z\_max*: 9, 19.  
*z\_min*: 9, 19.  
*z\_wall*: 9, 11.  
*zb\_max*: 9, 9.1, 9.2, 9.4, 14, 21, 21.1.  
*zb\_min*: 9, 9.1, 9.2, 9.4, 14, 21, 21.1.  
*zero*: 8, 9, 9.1, 9.2, 9.4, 12, 14, 15, 18, 19, 21, 21.4,  
 21.7, 21.8, 21.9, 22, 23, 24, 25, 26, 27, 33, 34, 35.  
*zi\_ix*: 23.  
*zi\_iy*: 23, 35.  
*zi\_iz*: 23.  
*zi\_ptr*: 16, 17, 23.  
*zn\_common*: 9, 23, 35.  
*zn\_exit*: 16, 23, 35.  
*zn\_index*: 16, 17, 23, 35.  
*zn\_plasma*: 16, 23, 35.

⟨ Allocations Initializations 9.1 ⟩ Used in section 9.  
⟨ C Functions 36, 37 ⟩ Used in section 7.1.  
⟨ DG Polygon Keyword 21.4 ⟩ Used in section 21.  
⟨ Edge Keyword 21.3 ⟩ Used in section 21.  
⟨ End Prep 9.4 ⟩ Used in section 9.  
⟨ Functions and subroutines 8, 9, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35 ⟩ Used in section 7.1.  
⟨ Intelligent Edge Processing 21.6 ⟩ Used in section 21.  
⟨ Intelligent Edge Specification 21.5 ⟩ Used in section 21.  
⟨ Memory allocation interface 0 ⟩ Used in sections 35 and 9.  
⟨ New Polygon 14 ⟩ Used in section 9.  
⟨ Outer Keyword 21.1 ⟩ Used in section 21.  
⟨ Process Closed iedge Polygon 21.7 ⟩ Used in section 21.6.  
⟨ Process DG File 10 ⟩ Used in section 9.  
⟨ Process Single Mesh Point 21.8 ⟩ Used in section 21.6.  
⟨ Process and Match Up Mesh Edge 21.9 ⟩ Used in section 21.6.  
⟨ Read Wall File 11 ⟩ Used in section 9.  
⟨ Set Bounds 9.2 ⟩ Used in section 9.  
⟨ Set Mesh Elements 13 ⟩ Used in section 9.4.  
⟨ Set Mesh Polygons 12 ⟩ Used in section 9.4.  
⟨ Set Symmetry 9.3 ⟩ Used in section 9.  
⟨ Setup End Zones 15 ⟩ Used in section 9.  
⟨ Setup Sector Diagnostics 17 ⟩ Used in section 9.  
⟨ Setup Toroidally Facing Sectors 16 ⟩ Used in section 9.  
⟨ Variable Keyword 22.1 ⟩ Used in section 22.  
⟨ Wall Keyword 21.2 ⟩ Used in section 21.

**COMMAND LINE:** "fweave -f -i! -W[ -ybs15000 -ykw800 -ytw40000 -j -n/  
/Users/dstotler/degas2/src/definegeometry2d.web".

**WEB FILE:** "/Users/dstotler/degas2/src/definegeometry2d.web".

**CHANGE FILE:** (none).

**GLOBAL LANGUAGE:** FORTRAN.